Scaling Concolic Execution of Binary Programs for Security Applications

Pongsin Poosankam

CMU-CS-13-112
August 2013

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was based upon work partially supported by the National Science Foundation under Grants No. 0311808, 0832943, 0448452, 0842694, 0627511, 0842695, 0831501, 0424422, and CCF-0424422, by the Air Force Office of Scientific Research under Grant No. 22178970-4170, by the Air Force Research Laboratory under Grant No. P010071555, by the Army Research Office under grant No. DAAD19-02-1-0389, by the Office of Naval Research under MURI Grant No. N000140911081, and by the MURI program under AFOSR Grants No. FA9550-08-1-0352 and FA9550-09-1-0539. Pongsin Poosankam was partially supported by the Royal Thai Government scholarship.

1. REPORT DATE **AUG 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**
4. TITLE AND SUBTITLE **Scaling Concolic Execution of Binary Programs for Security Applications**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		

14. ABSTRACT

Concolic execution is a technique for program analysis that makes the values of certain inputs symbolic, symbolically executes a program?s code, and computes a symbolic logical formula to represent a desired behavior of the program under analysis. The computed formula is then solved by a decision procedure to determine whether the desired behavior is feasible and, if so provide an example program input that satisfies the formula. Concolic execution and similar techniques have widely been applied to a variety of securityrelated applications including automatic test input generation, vulnerability discovery, exploit generation, signature generation, protocol reverse engineering and detecting deviations between software implementations. Although there has been a great success in applying it to various securityrelated applications, a basic implementation of concolic execution only works well on small programs and scaling it to real-world binary programs is difficult. One reason is that programs often contain certain code constructs that are difficult to reason about directly such as loops and encoding functions. Another reason is that the number of symbolic formulas grows drastically in proportion to the size of the program being analyzed. These observations led us to develop three scaling techniques for concolic execution. The first scaling technique, loop-extended concolic execution, focuses on improving the efficiency of concolic execution when analyzing program portions that involve loops. The second technique, decomposition and re-stitching of concolic execution, addresses the issue that arose from the presence of encoding functions, which are difficult to reason about automatically. The third technique uses the state model of the program under analysis to guide concolic execution. Our techniques work on program binaries and do not require the presence of source code or debugging information in the binaries. We apply our scaled concolic execution to a variety of security-related applications. For each of our scaling techniques, we demonstrate that they significantly improve the performance and usability of automatic test input generation and vulnerability discovery, which are previously known applications of concolic execution. We also study unexplored applications of concolic execution in security-related problems such as malware genealogy and protocol model inference.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**146**	

Keywords: BitBlaze, BitFuzz, binary program analysis, concolic execution, test input generation, vulnerability discovery

Abstract

Concolic execution is a technique for program analysis that makes the values of certain inputs symbolic, symbolically executes a program's code, and computes a symbolic logical formula to represent a desired behavior of the program under analysis. The computed formula is then solved by a decision procedure to determine whether the desired behavior is feasible and, if so, provide an example program input that satisfies the formula. Concolic execution and similar techniques have widely been applied to a variety of security-related applications including automatic test input generation, vulnerability discovery, exploit generation, signature generation, protocol reverse engineering, and detecting deviations between software implementations.

Although there has been a great success in applying it to various security-related applications, a basic implementation of concolic execution only works well on small programs and scaling it to real-world binary programs is difficult. One reason is that programs often contain certain code constructs that are difficult to reason about directly such as loops and encoding functions. Another reason is that the number of symbolic formulas grows drastically in proportion to the size of the program being analyzed.

These observations led us to develop three scaling techniques for concolic execution. The first scaling technique, loop-extended concolic execution, focuses on improving the efficiency of concolic execution when analyzing program portions that involve loops. The second technique, decomposition and re-stitching of concolic execution, addresses the issue that arose from the presence of *encoding functions*, which are difficult to reason about automatically. The third technique uses the state model of the program under analysis to guide concolic execution. Our techniques work on program binaries and do not require the presence of source code or debugging information in the binaries.

We apply our scaled concolic execution to a variety of security-related applications. For each of our scaling techniques, we demonstrate that they significantly improve the performance and usability of automatic test input generation and vulnerability discovery, which are previously known applications of concolic execution. We also study unexplored applications of concolic execution in security-related problems such as malware genealogy and protocol model inference.

Acknowledgments

Many people encouraged and supported the work described in this thesis. First and foremost, I thank my adviser and thesis committee chair Professor Dawn Song for her invaluable advice, guidance, and continuous support throughout my doctoral study. This work would have not been possible without her help.

I would like to gratefully acknowledge other four members of my thesis committee: Professor David Brumley, Professor Stephen McCamant, Professor Frank Pfenning, and Professor André Platzer. Though they were busy with other responsibilities, they graciously granted their time to work with me. Their insightful feedback and suggestions on how to improve this thesis and my research methodology in general are greatly appreciated.

Work described in this thesis is part of and utilizes parts of the BitBlaze infrastructure and I am indebted to all members of the BitBlaze team including Juan Caballero, Min Gyung Kang, Prateek Saxena, Heng Yin, Chia Yuan Cho, Steve Hanna, Zhenkai Liang, Lorenzo Martignoni, Damagoj Babic, Lenx Wei, Mathias Payer, Noah Johnson, Kevin Zhijie Chen, Daniel Reynaud, Ivan Jager, James Newsome, Dan Caselden, and Alex Bazhanyuk. I would also like to extend my gratitude to other colleagues who are co-authors in papers done during my doctoral study including Professor Avrim Blum, Professor Kevin Fu, Professor Vern Paxson, Professor Adrian Perrig, Professor Scott Shenker, Professor Ion Stoica, Petros Maniatis, Christian Kreibich, Li-Hao Liu, Shobha Venkataraman, Jiang Zheng, Rolf Rolles, Andres Molina-Markham, Edward XueJun Wu, Matei Zaharia, and Jun Han.

I would like to thank my parents for their unconditional love and support.

I would also like to express my appreciation to my friends in Pittsburgh and Berkeley as well as those in Thailand for always being there on good and bad days and for lending a helping hand whenever needed.

The final products of this thesis would not be possible without continuous help from the staffs at SCS and CSD, especially Deborah Cavlovich. I really appreciate your patience and your support. Thank you very much.

Portions of this thesis originate from work previously published in the proceedings of the 2009 International Symposium on Software Testing and Analysis (ISSTA) [99], the proceedings of the 2010 ACM Conference on Computer and Communication Security (CCS) [19], and the proceedings of the 2011 USENIX Security Symposium [26].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In this thesis, we develop techniques that scale concolic execution to broad classes of binary programs and apply it to a variety of security-related applications. We demonstrate that our scaling techniques significantly improve the performance and usability of previously known applications of concolic execution such as vulnerability discovery and signature generation. We then study further unexplored uses of concolic execution in security-related problems such as malware genealogy and protocol model inference.

Concolic execution is a technique for program analysis. By making the value of the program input symbolic, it symbolically executes a program's code and computes values for program variables in form of symbolic logical formulas. A computed formula is then provided to and can be solved by a decision procedure to determine whether it is possible for the corresponding variable to have some specific concrete value and what value the input must be, in the first place, for this to be feasible. Concolic execution and similar techniques have widely been applied to a variety of security-related applications. One of their most prevalent applications is *automatic test input generation* (also referred to as *program state-space exploration*) [21, 22, 23, 53, 54, 109]. In this application of concolic execution, a program is concretely executed once with some initial input. Then, a concolic execution engine can examine the branch conditions along the executed control-flow path and use a decision procedure to find an input that would reverse a branch condition from true to false or vice-versa. The process is repeated iteratively to discover more inputs that trigger new control-flow paths, and thus more program states to be tested. This technique is particularly useful for automatic generation of high-coverage test inputs and for software vulnerability discovery. Other security-related applications of concolic execution include vulnerability-based signature generation [13, 17], exploit generation [3, 64], protocol reverse engineering [18], and detecting deviations between software implementations [12].

Although there has been a great success in applying it to various security-related applications, a basic implementation of concolic execution only works well on small programs or on program procedures and scaling it to real-world binary programs is difficult. One reason is that programs often contain certain code constructs that are difficult to reason about directly. Common examples of such code constructs are loops and encoding functions such as decryption and checksum computation. Their existence results in symbolic formulas that tend to be large, complicated, and difficult to solve. Another reason is that the number of symbolic formulas grows drastically in proportion to the size of the program being analyzed. Without proper prioritization schemes, the overall approach becomes less and less efficient.

These observations led us to develop techniques that scale concolic execution to broad classes of binary programs. Our techniques work on program binaries and do not require the presence of source code or debugging information in the binaries. This provides us two key benefits toward security-related applications. First, it allows us to readily analyze a wide range of closed source software including commercial off-the-shelf (COTS) software and malware, which are already distributed in binary forms. In various cases, the users of COTS may want to analyze security properties of COTS because there is no guarantee that the software would be free of security flaws. In such situations, the ability to analyze the software from the binary directly is useful because the COTS developers may decide not to share the source code and related documentations. Similarly, in the case of malware, the identity of malware author is usually unknown at the time of its discovery and it thus leaves only the captured malware binaries as a starting point for security analysts. Second, the program binary is what gets executed and thus provides a more faithful representation of the program than the source code does. Semantics of the binary and the source code may vary slightly due to compiler errors and optimizations.

In this thesis, we are interested in programs that read and process some input and behave deterministically with respect to this input. Depending on the program and the goal of our analysis, the input can be anything including command line arguments, physical files, incoming network traffic, and the return values of system calls. Determinism provides us a guarantee that repeatedly executing a program with the same input always provides the same result. When performing concolic execution, determinism can be achieved on any programs by disabling run-time randomization, enforcing the same random seeds, or considering run-time non-deterministic variables as parts of the program input. In most cases, the presence or absence of randomization do not affect the results of our analysis.

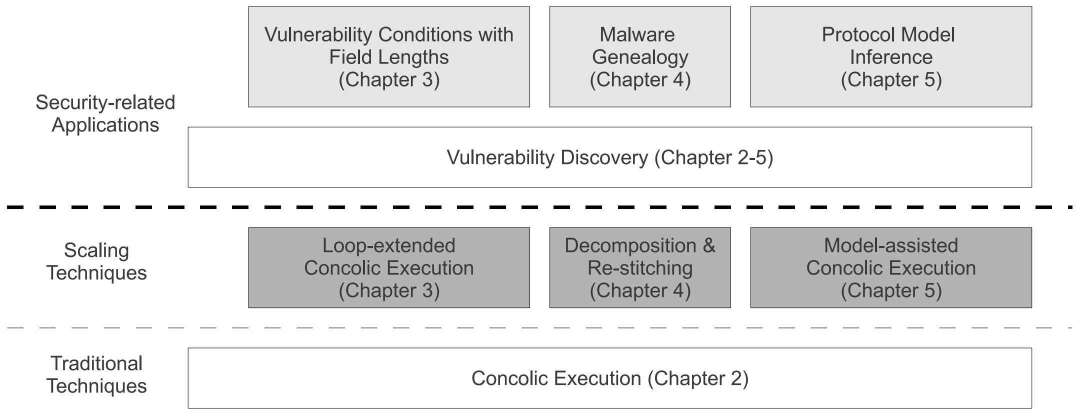

Vulnerability Conditions with Field Lengths (Chapter 3)
Malware Genealogy (Chapter 4)
Protocol Model Inference (Chapter 5)
Security-related Applications
Vulnerability Discovery (Chapter 2-5)
Scaling Techniques
Loop-extended Concolic Execution (Chapter 3)
Decomposition & Re-stitching (Chapter 4)
Model-assisted Concolic Execution (Chapter 5)
Traditional Techniques
Concolic Execution (Chapter 2)

Figure 1.1: Summary of Techniques and Applications.

This thesis consists of two main parts: techniques and applications. Figures 1.1 summarizes the techniques covered in this thesis and the security-related applications they enable. Unshaded boxes are the traditional technique and application already known prior to our work. Shaded boxes are novel techniques and novel applications first proposed in this thesis. In particular, we develop three techniques for scaling concolic execution. The first scaling technique, loop-extended concolic execution, focuses on improving the efficiency of concolic execution when analyzing program portions that involve loops. The second technique, decomposition and re-stitching of concolic execution, addresses the issue that arose from the presence of *encoding functions*, which are difficult to reason about automatically. The third technique, model-assisted concolic execution, uses the state model of the program under analysis to guide concolic execution. For each of our scaling techniques, we demonstrate that they significantly improve the performance and usability of automatic test input generation and vulnerability discovery, which are previously known applications of concolic execution.

We also study unexplored applications of concolic execution in security-related problems such as malware genealogy and protocol model inference. We show that our techniques enable some of these previously unexplored applications which were hindered by the scalability issue of concolic execution. In particular, loop-extended concolic execution allows us to describe vulnerability conditions in term of loop-related properties and lengths of input fields. A subsequent work [17] uses such vulnerability conditions to automatically generate vulnerability-based signatures that help detect malicious exploitation of the vulnerabilities. Another technique, decomposition and restitching, let us symbolically reason about the behaviors of malware, even when its communication is encrypted, and thus enable our study in malware genealogy. And by alternating our model-assisted concolic execution technique with L*, an existing online technique for inferring the high-level model of an application, we are able to improve the end results of both techniques and apply the combined technique to infer the protocol model of complex applications such as Samba and VNC with little manual intervention.

1.2 Main Contributions

In this section, we introduce the three scaling techniques we develop, and the security-related applications that the techniques enable.

1.2.1 Loop-extended Concolic Execution

Concolic execution and similar techniques have been widely used for finding and understanding software bugs, including security-relevant ones. To find software bugs, concolic execution first concretely executes a program, with some initial input, to create a path and then computes symbolic logical formulas to represent the branch conditions along the executed control-flow path. Through manipulation of the formulas, such as negation of a particular branch condition, a concolic execution engine can generate a new formula, which is then solved with a decision procedure. If a solution exists, the solution represents a new program input that shall take the program along a different control-flow path. The process is repeated iteratively to create inputs that cover different control-flow paths and may trigger hidden bugs in the program.

If a bug has already been observed and a sample buggy input has been given, concolic execution can be used to diagnose the bug. Again, one can concretely execute the program, with the buggy input being treated as symbolic, and examine the branch conditions that lead to the point of program failure. Other conditions on the value of input that trigger the bug can also be extracted using this approach. Previous researches [13, 17] have shown that these conditions, known as *vulnerability conditions*, are useful for automatically generating signatures to filter attacks, or to help a security analyst understand the vulnerability.

Problem Overview. Although concolic execution has been successfully used in the aforementioned applications, its existing approach is limited to examining behavior of a program one execution path at a time. This poses a significant challenge in making concolic execution scalable. When analyzing larger programs, the technique becomes less and less effective, largely due to the combinatorial explosion in the number of feasible execution paths.

This shortcoming becomes more apparent when concolic execution has to deal with loops, especially those whose number of loop iterations is controlled by the program input. In such cases, there are potentially many distinct execution paths to be examined individually, with each of those paths representing a different number of loop iterations being executed. Although the examined paths are distinct, their effects on the program's overall semantic may vary only slightly and a thorough exploration of such loops may be unnecessary. Simple heuristics, such as considering only execution paths with the same number of loop iterations as in the original program execution, can mitigate this combinatorial explosion problem. However, these heuristics will fail to expose bugs that are related to loops, such as buffer overflows, which are common and tend to be security-relevant.

In this thesis, we develop a new scaling technique, loop-extended concolic execution, which provides a middle ground for handling loops. Loop-extended concolic execution abstracts the effect of loops over all execution paths and summarizes them into a small and concise set of symbolic formulas. It does not suffer from the combinatorial explosion induced by the presence of loops but is yet able to reason about program behaviors related to the number of loop iterations.

Intuition and Approach. The traditional technique of concolic execution considers only direct data dependencies of program variables. When considering loops that iterate through program inputs, the formulas based on direct data dependency tend to be repetitive and high in number because each single loop iteration results in one distinct formula and thus make concolic execution unscalable. Our intuition is to replace these repetitive formulas with a more concise set of simplified formulas that are based on loop dependency instead. Thus, the goal of loop-extended concolic execution is to infer these loop-related formulas so that we can later perform symbolic reasoning on them. Specifically, the repetitive formulas are those from the loop exit conditions and the loop-related formulas are inferred from the loop invariants, which are properties that hold throughout the execution of the loop. Because the size of formula set is reduced from being bounded by the number of loop iteration to being bounded by the number of loop invariants, concolic execution is more scalable.

We need to express explicitly the inferred loop-related formulas so that we can pass them to the decision procedure. To achieve this, we introduce two new kinds of symbolic variables: loop trip counts and auxiliary variables. The loop-related formulas will be expressed in terms of existing symbolic variables as well as these new symbolic variables. A *loop trip count* represents the number of iterations each loop has been iterated at a specific point along the program execution. Thus, the number of loop trip counts will be the same as the number of loop occurrences. An *auxiliary variable* represents a property of the program input such as the length of an input field and an index to a field delimiter, which could affect the number of loop iterations during program execution. The number of auxiliary variables depends on how many variable-length fields there are in the program input.

Loop-extended concolic execution consists of two steps. In the first step, loop-extended concolic execution performs a one-pass forward symbolic analysis along a dynamic execution trace to determine dependencies of program variables on the loop trip counts. In particular, it searches for variables whose value is a linear function of one or more loop trip counts. If found, the linear function is the loop-related formula we seek and shall be added to the set of formulas that will be provided later to a decision procedure. Also, the formulas that were previously induced from the corresponding loop exit conditions shall be removed from the formula set.

In the second step, our technique heuristically analyzes how the program uses loops to access its input and searches for linear relationships between the auxiliary variables and the loop trip counts. These linear relationships allow our technique to express in symbolic formulas how loop-dependent variables relate to the lengths and counts of elements in the program input. These formulas are added to the set of formulas to be processed by a decision procedure. Because the auxiliary variables are symbolic in the formulas we provided to the decision procedure, the outputs we receive from the decision procedure will include the satisfying assignment for the auxiliary variables as well. Because the auxiliary variables represent properties of the program input such as the length of an input field and an index to a field delimiter, we can reconstruct new satisfying inputs of varied length.

Results. We have implemented our approach and applied it to discovery and subsequent diagnosis of buffer overflow vulnerabilities. We perform our analysis on a standard benchmark suite and on real-world software. The benchmark suite we use was previously published by researchers at the MIT Lincoln Laboratories [122]. It contains 14 samples inspired by vulnerabilities in open-source network servers. Starting from sample benign inputs, our tool discovers all known bugs denoted by the benchmark suite. Most of the bugs are found in just a few minutes. In addition to the known bugs, our tool also discovers a new bug in one benchmark.

As full-scale case studies, we test our approach on three real-world Windows and Linux programs which are known to have buffer overflow bugs. Our tool discovers four bugs in these programs in a few minutes. We also note that the computed symbolic formulas, which represents vulnerability conditions and thus corresponds to the generated buggy inputs, contains auxiliary variables that denote the lengths of input fields. They are more accurate and usable than those given in previous work [37], which lacks a notation to refer to the length of an input field.

To confirm the value of our approach, we count the number of concrete executions needed to discover each bug using our approach and that of using our implementation of the traditional (Section 2.4), and we compare them. In most cases, our approach significantly reduces the number of executions required to discover buffer overflow bugs.

Contributions.

- Loop-extended concolic execution: We introduce a new scaling technique for concolic execution that incorporates the semantics of loops into the traditional analysis. Our technique abstracts the effect of loops over all execution paths and summarizes them into a small and concise set of symbolic constraints. The technique works by introducing new symbolic variables that represent the number of loop iterations, using forward symbolic analysis to determine the generalized effects of these new symbolic variables on other program variables, and replacing loop-related constraints induced from individual execution paths with a small and concise set of generalized constraints.

- Improved vulnerability discovery: We implement our scaling technique and use it to find vulnerabilities in both a standard benchmark suite and three real-world programs. After generating only a handful of candidate inputs, our tool successfully discovers bugs in software. In most cases, our scaling technique significantly reduces the number of executions required to discover buffer overflow bugs.

- Vulnerability conditions with field lengths: By introducing auxiliary variables to represent features of an input grammar such as lengths and repetition counts, loop-extended concolic execution allows us to describe vulnerability conditions in term of loop-related properties and lengths of input fields. Subsequent work [17] uses such vulnerability conditions to automatically generate vulnerability-based signatures that help filter future attacks of known vulnerabilities.

1.2.2 Decomposition and Re-stitching of Encoding Functions

Problem Overview. Concolic execution naturally has issues when a program under analysis contains encoding functions. *Encoding functions* include tasks such as data decryption and encryption, data decompression and compression, and the computation of checksums and hash functions. When being symbolically reasoned about, encoding functions result in symbolic formulas that can be difficult to solve. This is not surprising because functions such as cryptographic hash functions are designed against finding any input that would provide the same hash as the original input. In particular, the problem we address is how to scale concolic execution to automatically generate test inputs for programs that use encoding functions.

Scaling concolic execution in the presence of encoding functions provides the first step toward answering another interesting security-related question: can we find and exploit vulnerabilities in malware? Although vulnerability discovery has long been an important task in software security, little research has addressed vulnerabilities in malware. Encoding functions are used widely in malware as well as in benign software. Many instances of malware such as trojans and botnets uses communication over encrypted channels to avoid being detected by network intrusion detection systems (NIDS). Improvement on the handling of encoding functions in concolic execution will greatly assist security researchers on malware analysis.

Intuition and Approach. To address the problem, we develop a novel technique that improves concolic execution on programs that use encoding functions. The intuition behind our technique is that it is possible to perform concolic execution without having to symbolically reason about encoding functions head-on. In particular, we avoid the complexity caused by encoding functions by identifying and bypassing them so that we can concentrate on performing symbolic reasoning on the rest of the program. Once having obtained partial results, we re-stitch them with the effect of the bypassed encoding functions using means other than symbolic reasoning to get a complete result. Because the technique is based on decomposing (factoring) the formulas induced by a program into subsets, solving only a subset, and then re-stitching the solutions back, we refer to our technique as *decomposition and re-stitching*.

Our technique starts by identifying encoding functions in the program execution and determining which form of decomposition is applicable for each encoding function. Other researchers have previously proposed algorithms to distinguish certain types of encoding functions from other functions in a program [114, 115] but they do not perfectly suit our purpose. We thus propose a new identification technique based on our intuition regarding the complexity caused by encoding functions: that the outputs of encoding functions contain a very high degree of mixtures from parts of the input, resulting in symbolic formulas that are difficult to solve.

To identify encoding functions, we perform a trace-based dependency analysis that is a general kind of dynamic tainting. Our analysis assigns an identifier to each input byte, and determines, for each value in an execution, which subset of input bytes it depends on. We call the size of the subset the byte's *taint degree*. If the taint degree of a byte is larger than a configurable threshold, we refer to it as high-taint-degree. Encoding functions are functions that produce high-taint-degree buffer as output.

After the encoding functions have been identified, we apply the traditional approach of concolic execution to generate the path constraint for the previously observed execution. A path constraint is a conjunction of smaller formulas induced from the instructions along the program execution and these formulas are annotated with the identifiers of the function they are induced from. We decompose the generated constraint to separate the conjoined formulas, single out those that come from encoding functions, conjoin the rest of the formulas to obtain a smaller constraint, and pass the constraint (now unrelated to encoding functions) to a solver. The constraint solution represents a partial input, which requires re-stitching to obtain the final input.

How to perform re-stitching depends on how the encoding function itself is used. For a function that decrypts or decompresses input data, we have to obtain its corresponding inverse function (i.e., encryption or compression) and supply this inverse function with a partial constraint solution (coming from the manipulation of decrypted input data) to get a complete (i.e., encrypted or compressed) solution. Finding the inverse function is possible by means of browsing and filtering through a list of candidate functions. For a function that computes and compares checksums, re-stitching is performed simply by concretely executing the function again on a partial constraint solution — a solution with everything but a correct checksum — to obtain a matching checksum that belongs to the final solution.

To determine how the encoding functions are used, our approach performs dynamic dependency analysis on the program execution trace to analyze how data is used inside and outside of the encoding functions and compares that against some pre-defined rules. For a decryption/decompression function, the data used by the function must never been used again later in the program. For a checksum function, its output must be used in a conditional check against a part of the program data that is disjoint from its input and that part of the program data must not be used elsewhere. According to our pre-defined rules, the usual usage pattern of performing a checksum verification on decrypted data will satisfy both rules and two-step decomposition and re-stitching can be performed. However, other usage patterns that combine multiple encoding functions may violate our pre-defined rules.

If an encoding function does not satisfy any pre-defined rules or if the inverse functions for functions like decryption and decompression are not present, decomposition and re-stitching is not possible. Our technique detects when such a case occurs and opts to perform concolic execution in a traditional brute-force manner.

Results. We have added an implementation of decomposition and re-stitching as an extension to our traditional concolic execution tool. We apply the tool, with and without the new scaling technique, to analyze four different samples of malware that extensively make use of encoding functions and demonstrate the improvement that results from the scaling technique. Under the traditional settings, the tool finds two bugs in two malware samples and runs out of memory on the other two families. Under the improved settings, the tool finds six bugs in all four malware samples and never runs out of memory. As a proof of concept, we successfully create an exploit attack that, once provided to the malware, triggers the hidden bug and uninstalls the malware from a host machine. To the best of our knowledge, our malware analysis is the first automated study of vulnerabilities in malware, though on a very small sample set.

We also retest the discovered buggy inputs on other malware instances from the same families and successfully trigger similar failure conditions in those instances as well. Some of these instances in the same family are first reportedly seen more than a few months apart. This implies that the bug, the encoding functions have not changed through time.

Contributions.

- Decomposition and re-stitching: We develop a technique that scales concolic execution to reason about programs that use encoding functions. The technique is based on decomposing the formulas induced by a program, solving only a subset, and then re-stitching the solutions into a complete result. The technique evidently improves the speed and reduces the memory usage of a concolic execution engine, making it more practical.

- Vulnerability discovery in malware: We use a concolic execution engine, coupled with our decomposition and re-stitching technique, to find bugs in malware instances that use encoding functions. We find six and publicly disclose five vulnerabilities in four malware instances that we analyzed. To the best of our knowledge, our malware analysis is the first automated study of vulnerabilities in malware.

- Malware genealogy: We also use the vulnerabilities we found to assist in the study of malware genealogy. We find that each of discovered vulnerabilities also appears in multiple variants in the same malware families and can be triggered using the same buggy input. Our study demonstrates that there are components in malware that tend to stay unchanged over time and thus could be used to identify the malware family to which an unknown suspicious binary belongs.

1.2.3 Model-assisted Concolic Execution

Problem Overview. Programs that maintain an ongoing interaction with its environment, like servers and web services, tend to get executed for a long period of time and thus performing state-space exploration (also referred to as automatic test input generation) using concolic execution on such programs can be time consuming. Nevertheless, we observe that the traditional approach of concolic execution can be improved if guided with the abstract model of the program under analysis. The abstract model could provide high-level information about the structure of program state space. By knowing how close (or how far) the analysis is from important states in the program, we shall be able to efficiently prioritize the overall process.

To simplify the problem of finding the right abstract model of the programs under analysis, we focus our technique on network applications that communicate with their environment through a protocol.

Approach. We propose an approach to combine an existing model inference technique with concolic execution. Our approach has three iterative steps. First, we use the existing technique to automatically infer an abstract finite-state model of a program's interaction with its environment. Second, we use the inferred model to guide concolic execution and to improve the state-space exploration. Third, if the exploration phase discovers new types of protocol messages, we refine the abstract model and repeat the process from the second step. We refer to our approach as *model-assisted concolic execution.*

We use *Mealy machines* to represent abstract protocol models. A Mealy machine is a finite state machine in which, at each particular state, an input from the environment determines what the output the model will emit and what state shall be transitioned into. Because the model is abstract, multiple distinct values of concrete program message (i.e., input or output) may be represented by the same abstract message. Our technique requires that the user provides an output message abstraction function. An *abstraction function* is a function that finds an abstract message which corresponds to a given concrete message.

Unlike one of our own prior work [27], our technique does not require an input message abstraction function.

Our technique requires a set of sample input messages that can be fed to the observed application. These sample input messages may be obtained from the live traffic. In its first step, our technique uses an existing black box inference algorithm, called L^* [2], to infer an abstract finite-state model of a network application using the set of sample program input messages. In a nutshell, L^* systematically feeds the network application under analysis with combinatorial sequences of the sample input messages (a message may appear in a sequence more than once), observes the network traffic induced by each message sequence, and soundly constructs a finite-state model that matches the observed traffics. Naturally, having a bigger set of sample input messages will result in a more complete model but will also require longer time to process.

In the second step, our technique uses the inferred model to guide a traditional concolic execution engine to discover more program input messages. In particular, for each of the states found in the inferred model, we feed a sequence of input messages known to lead the program to that particular state and perform concolic execution from that point on for an allotted time.

Finally, our automated technique selectively adds some of the newly discovered input messages to the current set of known input messages and again provides the set to L^* to infer an improved abstract model. We then repeat the process from the second step. In each iteration, the inferred protocol model will become closer to a complete model. After some number of iterations, no new state will be found and the model is converged.

Results.

We extend our traditional concolic execution engine with our technique and perform experiments with servers from two well-known network protocols: RFB (commonly known as VNC) and SMB. In particular, we pick Vino VNC 2.26.1 as a representative server for the RFB protocol and Samba 3.3.4 for the SMB protocol. After a few iterations, our technique successfully infers protocol models of both servers. Our technique discovers all the input messages (i.e., message types) as described by the RFB protocol specification[1] and as shown in the Samba source code[2]. It also generates, for each server, a finite state machine that resembles what the ideal protocol model would be.

One of the most prevalent applications of concolic execution is vulnerability discovery and our technique finds seven vulnerabilities in Vino VNC 2.26.1, RealVNC 4.1.2, and

[1]http://www.realvnc.com/docs/rfbproto.pdf
[2]http://www.samba.org

Samba 3.3.4 within 2.5 hours of concolic execution per state. We use the RFB model inferred by the analysis of Vino VNC to guide concolic execution of RealVNC. Our intuition is that RealVNC is another variant of servers that implements RFB protocol and thus should respect the same protocol model as Vino VNC does.

To confirm the value of our approach, we also compare the results of our vulnerability discovery against that of our implementation of the traditional approach (Section 2.4). When running for the same amount of wall clock time, a traditional concolic execution engine discovers only one of seven vulnerabilities found by our new technique. We also illustrate that our new technique is superior to the traditional approach in reaching deep states of the inferred protocol.

Contributions.

- Model-assisted concolic execution: We develop an iterative process of combining concolic execution with knowledge of an abstract model of the program under analysis. Our technique helps scale concolic execution to large network applications that communicate with their environment through some protocols. A complete protocol model is not our prerequisite because our technique can iteratively infer and refine an abstract model that represents the high-level logic of the network applications being analyzed.

- Vulnerability discovery in network servers: Our tool discovers seven vulnerabilities (four of which are new) in four applications that we analyze. We also show that our technique performs faster and deeper state-space exploration than the traditional approach.

- Protocol model inference: Given an output abstraction function, our approach iteratively infers and refines a model of the protocol as implemented by a program. Unlike the prior work, it requires no input abstraction function, which is usually not trivial and must be provided by the end users. Thus, our work contributes toward a more automated approach for reverse-engineering protocols.

1.3 Organization of the Thesis

The rest of this thesis is organized into three parts. In the first part, we discuss the traditional approach of concolic execution and its prevalent application of automatic test input generation. In Chapter 2, we discuss our implementation of technique and a sample case study of how we use the tool to discover a new vulnerability in a commercial software product.

In the second part, we discuss the shortcomings of traditional concolic execution, detail our techniques for addressing the issues, and describe unexplored applications of concolic execution that our new techniques enable. This part consists of three chapters. Chapter 3 describes loop-extended concolic execution, a technique that focuses on improving the efficiency of concolic execution when analyzing program portions that involve loops, and its application to vulnerability discovery and diagnosis of buffer overflows. Chapter 4 discusses the technique of decomposition and re-stitching, which addresses issues that arise from the presence of encoding functions, and how we adapt the technique to assist in malware analysis. Chapter 5 details the technique of combining concolic execution and automatic protocol model inference to improve both automatic test input generation and protocol reverse-engineering.

In the final part, Chapter 6 provides further discussion on the potential integration of our proposed techniques and the conclusion remarks.

Chapter 2

Concolic Execution of Binary Programs

2.1 Introduction

Designing secure systems is an exceptionally difficult task. Even a single bug hidden in an inopportune place can create catastrophic security holes. Considering the size of modern software systems, searching for and exterminating all the hidden bugs is a daunting task. Thus, development of tools and techniques that help reducing the severity of these security holes is of critical importance.

Testing plays an important role in finding bugs in a software product. Unlike static code analysis techniques like code review, inspection, and proof of correctness, testing involves monitoring actual program execution in hope of observing unexpected behaviors (e.g., program crashing or program prematurely terminated) which imply the existence of bugs. Programs under test are executed multiple times with different input values. Because the entire input set of programs tend to be too large to exhaustively test, usually only a subset are selected. Choosing a good subset contributes significantly to the effectiveness of a test process and can be done manually by test experts who have enough understanding of the program under test. However, such manual selection of test inputs is expensive and error-prone. Thus, various approaches to automatic test input generation/selection have been developed and adopted, ranging from random selection and heuristic-based selection to control flow-based selection and data flow-based selection [9, 80, 108].

Recently, concolic execution and other related techniques [65] have been popular for automatically generating test inputs for software systems [21, 22, 49, 53, 54, 82, 102, 113]. Concolic execution is a combination of concrete execution and symbolic reasoning. In its first step, it concretely executes a program under test with some initial input to create a concrete path. It then considers the seen input as symbolic and computes symbolic logical formulas to represent the branch conditions along the executed control-flow path. Through manipulation of the formulas, such as negation of a particular branch condition, it crafts new formulas, which can then be solved by a decision procedure. If a solution exists, the solution represents a new program input that shall take the program along a different control-flow path. From an instance of concrete execution, multiple new program inputs may get generated. The whole process is repeated with these new program inputs to generate even more program inputs that cover more unseen control-flow paths. By systematically exploring the program state space and generating one program input for each unique control-flow path, concolic execution ensures that the generated test inputs are not redundant and thus contributes to the area of efficient software testing.

In addition to automatic test input generation, concolic execution has been used in other areas as well. The application most related to test input generation is vulnerability-based signature generation [13, 17]. When a vulnerability is found (e.g., a crash discovered via automatic test input generation) and is reproducible, one can use concolic execution to perform symbolic reasoning along a faulty execution path and thus can obtain symbolic predicate on the program input that would lead the program along the same path and trigger the same vulnerability. This symbolic predicate can be used as a signature for detecting and protecting against malicious inputs that may potential harm the system. Because the symbolic predicate is not restricted to any particular concrete input, this type of vulnerability-based signature has zero false positive. In addition to automatic test input generation and vulnerability-based signature generation, other known security-related applications of concolic execution and similar techniques include exploit generation [3, 25, 64], protocol reverse engineering [18], and detecting deviations between software implementations [12].

We build BitFuzz, a trace-based concolic execution system, on top of the BitBlaze [105] platform for binary analysis. We successfully use BitFuzz for automatic test input generation and for finding bugs in software. BitFuzz tests a variety of Windows and Linux programs without the need of source code. The list of software we have tested includes commercial software like Cardiac Science G3 AEDUpdate software [57], server applications such as RealVNC[1] and Samba[2], and malware such as Zbot [40] and MegaD [77]. The standard version of BitFuzz implements the traditional approach of concolic execution. For many other software products, test input generation with the traditional approach alone does not yield good results. We additionally applied scaling techniques we develop to make it more practical. Thus, we discuss our test results on those software products in Chapter 3-5, where we also discuss our scaling techniques.

[1]http://www.realvnc.com
[2]http://www.samba.org

```
1   void process_command (char* msg) {
2     char num;
3     if (msg[0] % 8 == 0)
4       num = msg[0] - 8;
5     else
6       num = msg[0];
7     if (num == 'P')         // ASCII of P = 80
8       if (msg[1] == 'Y')    // ASCII of Y = 89
9         abort();      // represent a point of failure
10    return;
11  }
```

Figure 2.1: A Running Example.

2.2 Concolic Execution for Test Input Generation

In this section, we detail the steps taken by our traditional concolic execution tool to automatically generate test inputs for a given program. For simplicity, we describe the algorithm by example, leaving formal details of concolic execution to other literature in this area [53, 101].

Figure 2.1 contains a short C function. We use this function as a running example to illustrate how to perform concolic execution for automatic test input generation, treating its string argument as symbolic. For a conditional statement, we call the executions from the conditional statement line to the first line in the true block and the false block *a pair of branches*. Thus, in the example function, conditional statements are on line 3, 7, and 8, and pairs of branches are $(3 \rightarrow 4, 3 \rightarrow 6)$, $(7 \rightarrow 8, 7 \rightarrow 10)$, and $(8 \rightarrow 9, 8 \rightarrow 10)$.

Concolic execution performs symbolic reasoning on concrete execution paths and thus requires concrete inputs. Suppose that we have an initial input string "AB" (ASCII code = 65 and 66) and that concolic execution denotes the input as a string of symbolic characters $\mathcal{S}_0\mathcal{S}_1$. The concrete execution path with respect to this input will be $2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 10$. The full path predicate, which is the conjunction of conditions derived from statements along the execution path is $\neg(\mathcal{S}_0 \equiv 0 \bmod 8) \wedge (num = \mathcal{S}_0) \wedge \neg(num = 80)$. As you can see, the condition derived from a false branch (i.e., Line 3 and 7) is of the form $\neg P$ when P is the condition shown in a conditional statement. For the same conditional statement, a true branch will simply give a condition P. An assignment statement gives an equality condition in the same way Line 6 gives the condition $num = \mathcal{S}_0$ as shown in the path predicate.

For each condition derived from a conditional statement (known as *branch condition*), concolic execution modifies the path predicate by negating the branch condition and removing the conditions that come from subsequent statements. In our running example, the initial path predicate has two branch conditions and thus there will be two modified predicates: $\neg\neg(\mathcal{S}_0 \equiv 0 \bmod 8)$ and $\neg(\mathcal{S}_0 \equiv 0 \bmod 8) \wedge (num = \mathcal{S}_0) \wedge \neg\neg(num = 80)$. Each of the modified predicates are given to a constraint solver to solve for a sample input that would lead the execution along a different execution path, which is similar to the original execution path from the beginning down to the branch with the negated condition, if feasible. In this case, the first modified predicate is feasible. The satisfying input for this predicate is any byte string of which the first byte is a multiple of 8. Although there are many possible satisfying inputs, the constraint solver will only give one example of them. We will assume that the constraint solver provides us a satisfying input string "PB" (ASCII code = 80 and 66) for this predicate. The second predicate is not feasible because its first clause and third clause are in conflict (i.e., 80 is a multiple of 8). Thus, the constraint solver will tell us it is infeasible and nothing is to be done for this predicate.

The process is repeated. A concrete execution of the function with the newly generated input string "PB" will be along the path $2 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 10$. The full path predicate is $(\mathcal{S}_0 \equiv 0 \bmod 8) \wedge (num = \mathcal{S}_0 - 8) \wedge \neg(num = 80)$. Because both branches of the first conditional statement (Line 3) have been executed, we do not need to generate a modified predicate for that particular branch. Thus, concolic execution generates only one modified predicate with respect to the second conditional statement (Line 7), $(\mathcal{S}_0 \equiv 0 \bmod 8) \wedge (num = \mathcal{S}_0 - 8) \wedge \neg\neg(num = 80)$. The predicate is satisfiable and we will assume that the constraint solver provides us a satisfying input string "XB" (ASCII code = 88 and 66).

The process is repeated again. A concrete execution of the function with the new input string "XB" will be along the path $2 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 10$. The full path predicate is $(\mathcal{S}_0 \equiv 0 \bmod 8) \wedge (num = \mathcal{S}_0 - 8) \wedge (num = 80) \wedge \neg(\mathcal{S}_1 = 89)$. Because all branches of the first and the second conditional statements (Line 3 and 7) have been executed, only one modified predicate is generated. The predicate is with respect to the third conditional statement (Line 8) and it is $(\mathcal{S}_0 \equiv 0 \bmod 8) \wedge (num = \mathcal{S}_0 - 8) \wedge (num = 80) \wedge \neg\neg(\mathcal{S}_1 = 89)$. The predicate is satisfiable and we will assume that the constraint solver provides us a satisfying input string "XY" (ASCII code = 88 and 89).

When the function is executed with the new input, a hidden point of failure is reached and a bug is found. The path is $2 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9$. Because all branches of all conditional statements along the path have been executed, neither a new predicate nor a new input is generated.

As the function has been executed with the initial input and all other newly generated inputs, the automatic test input generation process is complete. The technique finds a concise set of test inputs that comprises four strings “AB”, “PB”, “XB”, and “XY”. The input string “XY” triggers a hidden failure in the function. This string is the only faulty two-byte string. Random test input generation, which is another automatic technique, will have a 2^{-16} success rate per each randomly selected input, to find the hidden failure. Thus, concolic execution is much better in term of the quality of test inputs generated and is a more favorable approach.

Although we uses a C function as a running example, our implementation of concolic execution works with binary programs and its algorithm is roughly the same as described in this walk-through.

2.3 Other Known Applications of Concolic Execution

In addition to automatic test input generation, concolic execution and related techniques have also been applied to other security-related areas as well. In this section, we briefly describe those applications.

Signature generation. Signature-based input filtering tests program inputs against a set of malicious input signatures and raises a warning when they are matched. It provides a fast and important means to protect a computer when patches for software vulnerabilities are not yet available or have not yet been applied. The key challenge to signature-based defense is to automatically generate small and efficient signatures that have both few false positives and few false negatives. One approach to this problem is to generate exploit-based signatures by extracting patterns that appeared in the observed exploits [70, 87, 111]. However, signatures generated by this approach can have high false positive and negative rates, especially when the exploits have many polymorphic variants.

Another approach to signature generation is to construct vulnerability-based signatures [13, 17] by analyzing the vulnerable program and the actual conditions needed to exploit the vulnerability. Given a vulnerable program and a point of vulnerability, this approach uses concolic execution to generate an input predicate that drives the program to the point of vulnerability. The generated signature is simply a conjunction of this predicate and the condition on the program input that exploits the vulnerability. Although vulnerability-based signatures may consist of complex constraints and may take a longer time to match against compared to regular expression-based signatures, they are more applicable because they have no false positive.

Protocol reverse engineering. One goal of protocol reverse engineering is to automati-

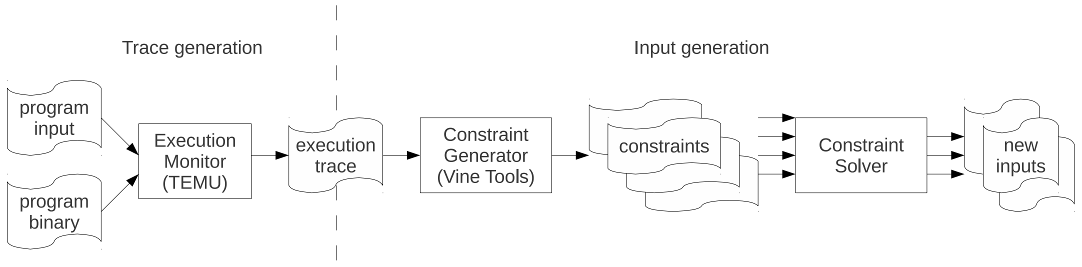


Figure 2.2: Overview of How BitFuzz Utilizes the Existing Components

cally extract the format of protocol input messages. Techniques for input format extraction can be classified into techniques that analyze the patterns of the valid inputs [35, 68] and techniques that analyze how a program processes inputs [20, 36, 117]. The second kind of techniques performs better when it observes more processing of program inputs. Thus, it benefits greatly from a high coverage set of test inputs generated using concolic execution [18].

Deviation detection. Different implementations that handle the same message protocol usually contain deviations, which are differences in how they process the same inputs. Detecting these deviations is important for several applications [12, 98]. Because differences between implementations often imply that one of them has an error, deviation detection can be used to detect potential errors in one of the programs. It can also be used to generate fingerprints, which are inputs used to remotely identify which applications or operating system a host is running. Lastly, in the case of two versions of the same software, it can be used to incrementally generate new test cases for a regression test suite, by focusing on inputs that cause the software to behave differently after it has been updated. Given two binary implementations and a sample input of a specific known type, previous work [12] uses concolic execution and constraint solving to automatically generate new inputs that demonstrate deviations between implementations.

2.4 BitFuzz: Our Concolic Execution Engine

We implement our symbolic reasoning engine, called BitFuzz. BitFuzz is written in OCaml, Python, and C/C++, and runs on a Linux system. BitFuzz does not require source code of the program being analyzed. It analyzes Intel x86 binaries running in a Windows or Linux OS, and is extensible to analyze binaries in other architectures and other operating systems. BitFuzz symbolically reasons about a subject program to generate a high coverage set of inputs that exercise feasible program paths. To address the general scalability issue of concolic execution, we extend BitFuzz with scaling techniques which are discussed in Chapter 3-5.

As illustrated in Figure 2.2, BitFuzz is implemented using several existing components in the BitBlaze [105] platform for binary analysis. TEMU, the first existing component, is an extensible whole-system emulator. It is used to execute the subject program in its expected operating system environment (such as unmodified versions of Windows XP and Ubuntu Linux). The relevant inputs to the program are marked and tracked using dynamic taint analysis: they can come from the virtual keyboard, from a disk file, from a network message, or from any specially designated API routine. BitFuzz uses TEMU to observe the instructions that operate on the inputs, and to save them to an *execution trace file* that also records the inputs and their arguments. This trace file is parsed using the Vine toolkit, the second existing component, which comprises an intermediate language and analysis library that represents the precise semantics of the original Intel x86 machine instructions using a small set of more general and simpler operations. BitFuzz uses the Vine toolkit to analyze and extract the symbolic path predicate from the intermediate representation of the trace. Finally, it queries a constraint solver, the third existing component, to solve whether the symbolic path predicate is satisfiable and to provide an example input whose assignment would satisfy the path predicate. The experiments in this thesis use STP [48], a complete decision procedure incorporating the theories of arrays and bit-vectors, as a constraint solver for BitFuzz.

In the context of automatic test input generation, BitFuzz follows the technique describe in Section 2.2. It modifies the path predicate by negating one of the branch conditions in depth-first order and removing the conditions that come from subsequent branches. It then queries the decision procedure to solve the modified predicate for an input that will direct the program to a different path starting at the modified branch. BitFuzz repeats this process to generate inputs that explore various program paths. It negates each symbolic branch condition that appears in a trace to generate more inputs. And for each unique input, BitFuzz reruns TEMU to collect the corresponding trace.

Rather than being limited to a simple back-and-forth alternation, BitFuzz maintains two priority queues, one for candidate inputs and another for collected execution traces, as illustrated in Figure 2.3. Each input gives a trace and each trace can yield multiple new inputs. The traces that visit a larger number of new basic blocks, unexplored by the prior traces, have higher priority. The inputs generated by negating branch conditions inside newly discovered basic blocks also have higher priority. Similar to the prioritization

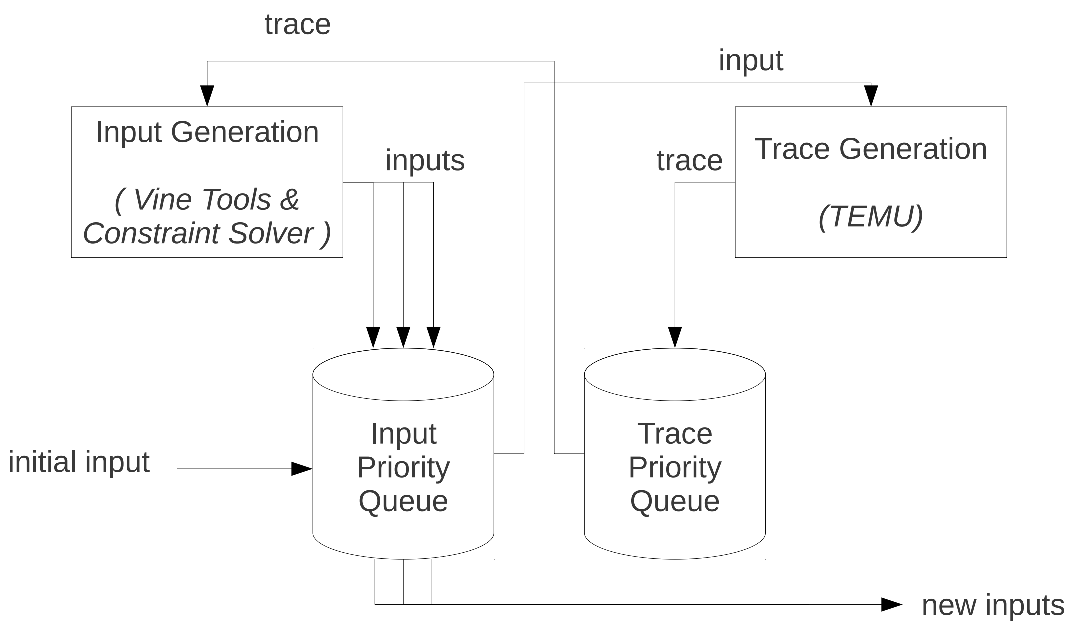


Figure 2.3: Architecture of BitFuzz

schemes described in the related work [15, 54], our priority queues are biased toward finding new program paths.

Vulnerability detection. BitFuzz supports several techniques for vulnerability detection and reports any inputs flagged by these techniques. It detects program termination and invalid memory access exceptions. Executions that exceed a timeout are flagged as potential infinite loops. It also uses TEMU's taint propagation module to identify whether the input (e.g., network data) is used in the program counter or in the size parameter of a memory allocation.

Distributed environment. BitFuzz is designed to be scalable and distributed; it can be run on a single machine and on multiple machines that share resources. We have successfully run a parallel distributed version of BitFuzz across multiple machines under DETER Security testbed [8]. We have also setup BitFuzz on Amazon EC2 cloud with a goal of letting users try whether the approach of concolic execution is applicable to their security analysis needs.

Chapter 3

Loop-extended Concolic Execution

3.1 Introduction

A key limitation of traditional concolic execution (or TCE for short) is that it interacts poorly with loops, a common programming construct. Specifically, it generalizes a program execution only to a set of executions that follow exactly the same number of loop iterations for each loop as in the original concrete execution (with exception of the path predicate in which one of the loop exit conditions is negated). For instance when traditional concolic execution is applied to test case generation to increase coverage, it will be unable (in one iteration) to generate an input that forces execution down a different branch than in the original execution, if taking that branch is only feasible with a different number of loop iterations. In other words, in traditional concolic execution, the values of a symbolic variable reflect only the data dependencies on the symbolic inputs; control dependencies, including loop dependencies, are ignored.

We propose *loop-extended concolic execution* (or LECE for short), which generalizes from a concrete execution to a set of program executions which may contain a different number of iterations for each loop as in the original execution. In loop-extended concolic execution, in addition to the data dependencies on inputs, the value of a symbolic variable also captures certain loop dependent effects.

At a high level, our approach works by introducing new symbolic variables to represent the number of times each loop in the program has executed. In addition to maintaining the data dependencies of program state variables on inputs as in TCE, LECE performs a more detailed analysis to identify loop-dependent variables, for instance finding variables whose value is a linear function of one or more loop execution counts. It also relates loop execution counts to features of the program input, introducing auxiliary variables to capture the lengths and repetition counts of fields in a known input grammar. Together, these constraints allow LECE to additionally express how loop-dependent variables relate to the lengths and counts of elements in the program input.

Loop-extended concolic execution can make bug-finding tools more effective and allow test-case generation to reach high coverage more quickly. Capturing more program logic in symbolic constraints allows LECE to reason about loop-related constraints with a decision procedure, rather than requiring iterative undirected search as with TCE.

The power of LECE is crucial for several important applications. As sample applications, we use loop-extended concolic execution to discover and diagnose buffer-overflow vulnerabilities, one of the most important classes of software errors that allow attackers to subvert programs and systems. Intuitively, LECE is powerful enough to express the effect of varying features of the input, such as number of fields or their lengths (which, in turn, affect the loop iteration counts), on program variables in a single step. This allows new vulnerabilities to be discovered using many fewer iterations than traditional concolic execution. In addition, for a known vulnerability, our techniques are useful to diagnose a set of general conditions under which it may be exploited. These conditions are useful for understanding the vulnerability, testing for it, fixing it, and blocking attacks targeting it [13, 31, 32, 37, 47, 54, 119, 120, 122].

Because concolic execution is often used in security-related applications such as this one, it is important that it works well for binary programs for which source code is not available. Our algorithms are designed with this constraint in mind. They overcome some of the challenges inherent in targeting binaries—such as recovering program structures like the boundaries of loops, which appear trivially in the original source.

We have built a full implementation of this technique, using a dynamic tool to collect program traces and an off-the-shelf decision procedure to simplify and solve constraints. Our tool discovers and diagnoses vulnerabilities in both a standard benchmark suite and three real-world programs on Windows and Linux. Our results show that LECE is practically effective, and confirms that the behavior of loops in real programs is often very regular.

3.2 Overview

In this section, we first motivate our approach with an example showing the limitation of traditional concolic execution, then give an overview of our technique of loop-extended concolic execution.

3.2.1 Motivation and Challenges

Using concolic execution to generalize over observed program behavior is a powerful technique because it combines the strengths of dynamic and static analysis. However, the core traditional concolic execution technique corresponds to an analysis of just one control-flow path in a program, which is a significant limitation in programs that contain loops. Next, we show this limitation with a specific example.

Consider a simplified example of a function in an HTTP server, shown in Figure 3.1, that processes HTTP GET requests. The program first checks that the request's method field has the value `GET` on line 9, and then proceeds to parse the URI and version fields into separate buffers on lines 12–16 and 18–22 respectively. It rejects this request if the version number is unsupported. Finally, it records the URI requested by the client and the version number in a comma separated string denoted by `msgbuf` on lines 26-30, which it subsequently logs by invoking `LogRequest` on line 32.

Readers may have already noticed that this code is vulnerable to a buffer overflow, but suppose we were attempting to check for such vulnerabilities using a traditional concolic execution technique. For instance, in the course of its exploration, such an iterative test generation tool might consider the program input `GET x y`. It will trace the execution of the program with this input, which causes the program to reach the error condition on line 24. In order to explore the rest of the function, the exploration tool needs to find a program input that passes the checks on line 23. However, a single path does not contain enough information to reason about the length check, because the `ver_len` variable is not directly dependent on any byte of the input: traditional concolic execution would not mark it as symbolic. At this point, testing tools based on concolic execution will usually attempt to explore other program paths, but without information from the first path to guide them, they can only choose further paths in an undirected fashion, such as by trying to take a different direction at one of the branches that occurred on the observed path. (Such tools treat the execution of a loop simply as a sequence of branches, one for each time the loop end test is executed.) For instance, a tool might determine that changing the last character of the input from a newline to `z` would cause the loop at line 18 to run for one additional iteration. A series of many such changes would be required before the version field was long enough to pass the check.

Similarly, consider the execution of the program on the normal program input `GET /index.html HTTP/1.1`. For this simple function, a single input already exercises a

```c
#define URI_DELIMITER ' '
#define VERSION_DELIMITER '\n'

void process_request(char * input)
{
  char URI[80], version[80], msgbuf[100];
  int ptr=0, uri_len=0, ver_len=0, i, j;

  if (strncmp input, "GET ", 4) != 0)
      fatal("Unsupported request");
  ptr = 4;
  while (input[ptr] != URI_DELIMITER) {
      if (uri_len < 80)
          URI[uri_len] = input[ptr];
      uri_len++;    ptr++;
  }
  ptr ++;
  while (input[ptr] != VERSION_DELIMITER) {
      if (ver_len < 80)
        version[ver_len] = input[ptr];
      ver_len++;    ptr++;
  }
  if (ver_len < 8 || version[5] != '1')
      fatal("Unsupported protocol version");

  for (i=0,ptr=0; i < uri_len; i++, ptr++)
      msgbuf[ptr] = URI[i];
  msgbuf[ptr++] = ',';
  for (j = 0; j < ver_len; j++, ptr++)
      msgbuf[ptr] = version[j];
  msgbuf[ptr++] = '\0';
  LogRequest(msgbuf);
}
```

Figure 3.1: A Simplified Example from an HTTP Server that Handles Requests of the Form: `"GET␣"` *URI* `"␣"` *Version* `"\n"`

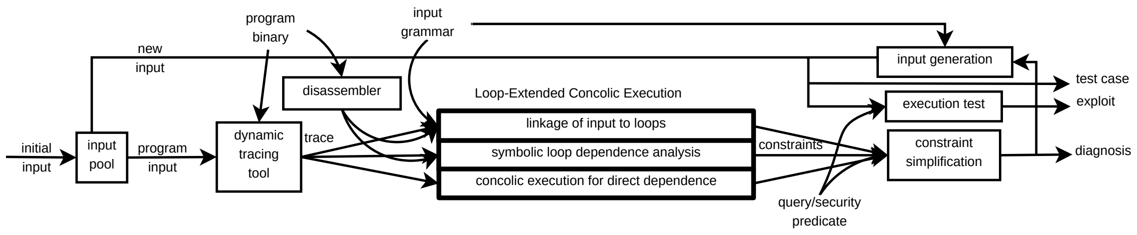


Figure 3.2: Overview of Our LECE Tool and Accessory Components. LECE, our main contribution, enhances concolic execution for directly input-dependent data values, as in traditional concolic execution, with symbolic analysis of the effects of loops (Section 3.3.1) and an analysis that links loops to the input fields they process (Section 3.3.2). Additional components, described in Sections 3.4 and 3.5, support LECE and particular applications such as detecting and diagnosing security bugs.

large proportion of the code (for instance, it executes all of the lines of non-error code in the figure). However, examining this single path is not enough to elucidate the relationship between the variable `ptr` and the input, because that relationship involves control dependencies.

3.2.2 Technique Overview

We propose a new type of concolic execution, *loop-extended concolic execution* or LECE, which captures the effects of more related program executions than just a single path (as in traditional concolic execution), by modeling the effects of loops.

Broadly, the goal of loop-extended concolic execution is to extend the symbolic expressions computed from a single execution by incorporating additional information reflecting the effects of loops that were executed. In traditional concolic execution, the values of variables are either concrete (i.e., constant, representing a value that does not directly depend on the symbolic input) or are represented by a symbolic expression (for instance, the sum of an input byte and a concrete value). However, some of the values considered concrete by traditional concolic execution are in fact indirectly dependent on the input because of loops. In loop-extended concolic execution, these values can also be represented symbolically, and variables whose values were already symbolic because of a direct input dependency can have a more general abstract value.

To make loop-extended concolic execution more tractable, we split the task into two parts by introducing a new class of symbolic variables, which we call *trip counts*. Each loop in the program has a trip count variable that represents the number of times the loop has executed at any moment. Then to obtain the relationship between a symbolic values and the program input, we separately obtain first the relationships between the symbolic values and one or more trip counts (in addition to their direct relationships with the input, as in traditional concolic execution), and then the relationships between the program's trip counts and the program input:

- **Step 1: Symbolic analysis of loop dependencies.** To determine dependencies on loop trip counts, we use a program analysis that maintains the trip counts as symbolic variables that are implicitly incremented for each new loop iteration, and then looks for relationships between those variables and others in the program. (This is done at the same time as the analysis tracking direct dependencies as in TCE, and the results combined in single symbolic expressions.) Specifically, we have found that looking for linear functions of the trip counts covers the most important loop dependent variables without excessive analysis cost. Traditional induction variable is one of such variable, as it depends on a particular loop that it resides in, but not vice versa. Our loop dependent variable may depend on trip counts of multiple loop occurrences in a program.

- **Step 2: Constraints linking the input grammar to loops.** Loops are often used when fields of the input are of variable length, such as character strings and sequences of data of the same type. Our approach takes advantage of this connection by using a grammar that specifies the inputs to the program, and matching loops with the parts of the input over which they operate. In particular, the approach introduces *auxiliary* input variables to capture features of the grammar such as lengths and repetition counts.

A summary of the components of our system is shown in Figure 3.2; the center box, LECE, represents the primary contribution of this research.

To summarize our approach, we now return to the example of Figure 3.1 and explain how loop-extended symbolic execution is more helpful to our vulnerability testing application.

1. In the first step, the symbolic loop dependence analysis expresses various program values in terms of four trip count symbolic variables TC_i, one for each loop i in the program. For instance, the value of the variable `ptr` at the end of execution is abstracted by the expression $TC_3 + TC_4 + 2$, and similarly `uri_len` $= TC_1$, `ver_len` $= TC_2$, `i` $= TC_3$, and `j` $= TC_4$. The path predicate is also maintained (as in traditional concolic execution). In this example, for instance, `i` $<$ `uri_len`

inside the third loop, while the negation holds after the loop has completed, and similarly for `j` and `ver_len`.

2. In the second step, we link the trip counts to auxiliary variables representing features of the input. In the running example, the execution counts of the first two loops are equal to the lengths of input fields: $TC_1 = \text{Length}(URI)$ and $TC_2 = \text{Length}(Version)$.

In the case of vulnerability checking, we would combine these symbolic constraints describing a class of program executions with the condition for a violation of the security policy. In this case, for instance, the array access on line 30 will fail if `ptr` ≥ 100. Then in the same way as in a traditional concolic execution approach, we can pass these conditions to a decision procedure to determine whether an exploit is possible, and if so, determine specific values for input variables that will trigger it. In this case, the decision procedure will report that an overflow is possible, specifically on an input for which $\text{Length}(URI) + \text{Length}(Version) \geq 99$.

Applying the approach to binaries. Because we wish to use these analysis techniques for security applications, it is an important practical consideration that they work on binary programs for which source code is not available. This adds further challenges for our approach: for instance, purely static analysis is more difficult on binaries because much of the structure that existed in the source code has been lost. (And of course, the real constraints we generate do not contain variable names, which we added in the example for readability.) It is in part for this reason that the concolic execution approach is valuable in the first place, so we choose algorithms to retain these benefits in our extension. For instance, even though the technique we use to infer linear relationships between variables is closely related to a sound static analysis approach, we do not limit it to finding relationships that could hold on all possible inputs. Instead, our goal is to combine static and dynamic analysis to produce results that cover as large as possible a range of inputs for which we can still produce useful results.

Use of an input grammar. Information that constrains the space of valid inputs to a program, in the form of a grammar or otherwise, is key to scaling input space exploration beyond the limits of brute-force exhaustive search. Previous research using concolic execution [17, 52, 74] demonstrates the benefit of using an input grammar for this purpose. In the application domains we target, suitable grammars are easily available, so we simply use them. However, for domains in which grammars are not already available, previous research shows how a grammar can be inferred [20, 71, 117]; such a system could easily be combined with ours.

3.3 Algorithms

In this section, we discuss the algorithmic details of the key steps in loop-extended concolic execution introduced in Section 3.2. Section 3.3.1 describes the analysis that identifies relationships between values of variables and numbers of loop iterations (step 1). Section 3.3.2 outlines techniques to capture the relationships between loops and the input, using auxiliary variables in the external specification of the input grammar (step 2).

The steps described below require accessory components to extract control flow graphs from binaries, make irreducible CFGs reducible, extract sizes of allocated objects, and parse input grammars. The details of these components, which form the preparation phase for steps outlined here, are given later in Section 3.5.

Loop detection. Our approach uses an existing static loop detection technique to identify all loops in the subject program. To achieve this, it uses an accessory component (to be discussed in Section 3.5) to obtain the control flow graph of the program. Then, it applies a static loop detection algorithm [84, pp. 191–197] to identify natural loops in the control flow graph by means of searching for back edges. A *back edge* $m \to n$ is an edge in the control flow graph whose terminal node n dominates its initial node m (i.e., every path from the entry of the control flow graph to the node m contains the node n). For each back edge $m \to n$ found, a corresponding *natural loop* is a subgraph whose node set includes n and any nodes that can reach m without visiting n, and whose edge set contains all the edges connecting nodes in the node set. The node n is called the *loop entry* of the natural loop. A natural loop has one loop entry but may have multiple back edges. A loop entry is unique to each natural loop; two loops that share the loop header are actually considered as a single natural loop.

For each natural loop, the algorithm gives us the address of each loop entry, the address pairs that compose each loop back edge (a loop may have multiple back edges), and the loop exit condition. Our approach then uses these addresses in a pass over the dynamic execution trace to detect the occurrences of loops in the execution and uses this information to assist the symbolic trip count analysis (Section 3.3.1). The loop exit conditions are used to determine the relationship between loops and the program inputs (Section 3.3.2) and to generate loop-extended symbolic constraints (Section 3.4.1).

We also consider using a dynamic approach to detect loop occurrences in the execution trace [66]. The approach requires less setup as it does not requires a control flow graph

$Trace$	::=	$Stmt^*$
$Stmt$	::=	$lhs \leftarrow e \mid$ `assert(`$cond$`)`
lhs	::=	$v \mid *v$
$cond$	::=	$e1 = e2 \mid e1 < e2 \mid \neg cond' \mid cond_1 \wedge cond_2 \mid cond_1 \vee cond_2$
e	::=	$v \mid *v \mid c \mid e_1 \circ e_2$ where $\circ \in \{+, -, *, /, \%, ...\}$, v is a variable, and c is a constant

Table 3.1: Syntax of an execution trace.

and works on any execution trace (e.g., can be used against packed malware). However, it is less reliable because it does not detect every kind of loops in the trace [66] and also requires heuristics to find the loop exit conditions. Because the knowledge of exit conditions is crucial to our overall approach, we choose the static loop detection approach over its dynamic counterpart.

3.3.1 Symbolic Analysis of Loop Dependencies

In order to generalize its description of computations that involve loops, our approach must determine the relationship between loop-dependent variables and the loops in which they are modified. To achieve this, our approach performs a one-pass forward symbolic analysis along a dynamic execution trace. Specifically, it searches for variables whose value is a linear function of *trip count* variables. Each of these trip count variables represents the number of iterations each loop executes.

There are two advantages of our dynamic analysis approach over a completely static one such as the induction variable identification approach commonly performed in compilers [1, pp. 687–688]. First, our approach keeps track of dependencies on loop execution counts even after the loop itself has finished, and combines dependencies on multiple loop occurrences. In contrast, a static induction variable analysis only reasons about induction variables with respect to one particular loop. Second, our approach is performed on a dynamic trace and thus does not require a conservative alias analysis, which is often a source of scalability challenges and/or imprecision in static analysis.

Analysis algorithm. The goal of our algorithm is to find the linear relationship between loop-dependent variables and the loops they depend on. For each loop occurrence l in the program, we introduce a symbolic trip count variable TC_l, which represents the number of iterations the loop has executed. The core of an abstract value in our analysis is a symbolic linear expression whose terms are trip counts, with integer scaling factors and an integer constant term. For instance, the abstract value $10 + 4 \cdot TC_1 + 2 \cdot TC_2$ would correspond to a variable initialized as 10, then incremented by 4 on each iteration of the first loop and by 2 on each iteration of the second loop. If a variable is loop-dependent, its abstract value will have at least one trip count term. Otherwise, it will be either a constant or undefined.

Our dynamic analysis algorithm for determining the linear relationship between variables and loops is shown in Figure 3.3 and Figure 3.4. An abstract value is stored in a record data type $abstrval$ (lines 1–5), which comprises an integer constant term c and integer vector v representing scaling factors of the trip count terms. For example, if the number of loop occurrences is 5, the abstract value $10 + 4 \cdot TC_1 + 2 \cdot TC_2$ would correspond to `<`c`=10,` v`=[4,2,0,0,0]>`. Thus, the length of the vector v is bounded by the number of loop occurrences in the trace. If an abstract value is not linear to the trip counts, a nil record is used.

The goal of our algorithm is to obtain an abstract store, which is a map of a variable (temporaries and machine registers in our machine-level trace) or a memory location to its abstract value. This map is represented in our algorithm by the data type $abstrmap$ (line 6). It allows us to determine the relationship between variables and loops at any point throughout the execution trace.

The main routine of our algorithm is `analyzeTrace` (lines 8–26). Given an execution trace (syntax shown in Table 3.1), `analyzeTrace` performs a pass over the trace to compute and update M, which is the abstract store at that particular point in the execution.

When propagating through a loop occurrence, `analyzeTrace` uses l, $P[l]$ to keep record of the abstract store at the end of the most recently completed loop iteration, or at the loop entry if we are within the first iteration (lines 14–17, 19–23). Upon reaching the end of each loop iteration, the recorded abstract store and the current abstract store are used to compute an abstract value for each variable in term of loop trip counts and to verify any abstract values computed in prior iterations (routine `joinIterations`). In particular, at the end of the first iteration, the scaling factor of the loop trip count term is computed as the change in the abstract value from when it was at the beginning of the iteration (lines 32–34). At the end of other iterations, we verify this same scaling factor against the ones computed in the prior iterations. If they do not match, it means that the variable is not loop dependent (lines 35–37). Once a variable is deemed not to be loop dependent, it stays that way under the loop under analysis is terminated (lines 31 and 38).

`analyzeTrace` also calls a routine `analyzeStmt` (the routine has two helper routines — `abstrEval` and `lookupAndStore`) to update the abstract store with the side effect of each assignment statement (lines 43–47); statement of other types are ignored (line 48–49). If the source of the assignment is a variable, a memory location, or a constant, its abstract value is directly inherited to the destination (lines 55–57). If the source is an arithmetic operation that involves a summation, a subtraction of two loop-dependent values, or a multiplication of a loop-dependent value and a constant, we compute a linear formula for the destination (lines 58–69). A multiplication of two loop-dependent values and other operations do not result in a loop-dependent value (lines 70–71).

For instance, consider the analysis of loop 3 on lines 26–27 of Figure 3.1. At the beginning of the loop, `ptr` has the abstract value <c=`0`, v=`[0,0,0]`> and the abstract store is recorded. Then, `ptr` is incremented and thus have the abstract value <c=`1`, v=`[0,0,0]`>. At the end of the first iteration, the recorded abstract value and the current abstract value are used to obtain a new abstract value <c=`0`, v=`[0,0,1]`>, which will pass the verification at the end of each subsequent iteration. The effect of the increments on lines 28 and 31 and loop 4 on lines 30–31 are analyzed in a similar way, giving a final abstract value for `ptr` of <c=`2`, v=`[0,0,1,1]`>, which corresponds to $2 + TC_3 + TC_4$.

Our approach bypasses the issues that arises from conservative alias analysis by distinguishing memory locations using the concrete addresses observed in the execution trace. When a symbolic value is used as a memory address (e.g., indexing an array), our approach use the concrete address value, as is common in traditional concolic execution (lines 46 and 56).

Our approach also aims to identify as many as possible abstract values that are linear expressions, so that they are available for our subsequent analysis. To achieve this, we allow our tool to convert non-linear abstract values (represented by nil record) to the constant value representing the value the variable had in the concrete trace at the point. This is similar in effect to removing from consideration all the executions on which that variable had any other value, though less drastic because those executions can still contribute to the generality of other abstract values. Given that there is a limit to the amount of generality our abstract values can represent, this conversion reflects a judgment that it is more valuable for them to abstract over variation that occurs close to the point where they are queried. For instance, if the combined effect of two nested loops is nonlinear, our analysis will retain the dependence on the inner loop's trip count.

Theoretically, it is not clear when the best points to convert an abstract value in this way would be: for instance, delaying a conversion at one program point might remove the need to convert another value later. However, we have had good results by performing the conversion eagerly just before a non-linear abstract value would otherwise propagate. Specifically, it is performed at the end of each loop iteration after the abstract value has been computed or has been verified (lines 37–38).

```
type $abstrval$ = record {c: int, v: int[]};
  /* Represent abstract values:
     c+v[0]$\cdot TC_1$+v[1]$\cdot TC_2$+...+v[$N$-1]$\cdot TC_N$, when v has size $N$.
     c            , when v is nil.
     NON-LINEAR  , when the record is nil. */
type $abstrmap$ = map {string -> $abstrval$};

procedure analyzeTrace($trace$ : $Trace$) returns $abstrmap$ {
  $M$ : $abstrmap$ = {};
  $P$ : $abstrmap$[loop occurrences in the trace];
  $l$, $i$ : int;

  for each $stmt$ in $trace$ {
    if ($stmt$ is a loop entry) {
      $l$ = the loop that starts at $stmt$;
      $P$[$l$] = $M$.copy;
    }
    $M$ = analyzeStmt($stmt$, $M$);
    if ($stmt$ is an end of some loop iteration) {
      $l$ = the loop whose one of its iterations ends at $stmt$;
      $i$ = number of complete iterations of the loop $l$ at $stmt$;
      joinIterations($l$, $i$, $P$[$l$], $M$);  /* $M$ modified */
      $P$[$l$] = $M$.copy;
    }
  }
}

procedure joinIterations($l$, $i$ : int, $P$, $M$ : $abstrmap$) {
  for each key $v$ in $M$ {
    $p$ = $P$[$v$];  $m$ = $M$[$v$];
    if ($p$ is not nil && $m$ is not nil && $p$.v == $m$.v) {
      if ($i$ == 1) {    /* Infer an abstract value */
        $M$[$v$].c = $p$.c;
        $M$[$v$].v[l] = $m$.c - $p$.c;
      } else if ($p$.v[l] == $m$.c - $p$.c) {
        /* The existing abstract value is valid. Do nothing. */
      } else $M$[$v$] = < c = concreteValue($v$) , v = nil >;
    } else $M$[$v$] = < c = concreteValue($v$) , v = nil >;
  }
}
```

Figure 3.3: First Part of the Pseudocode for Our Symbolic Analysis.

```
procedure analyzeStmt($stmt$ : $Stmt$, $M$ : $abstrmap$) returns $abstrmap$ {
  match $stmt$ with
    $lhs \leftarrow e$:
      match $lhs$ with
        $v$:  $M[v]$ = abstrEval($e$, $M$);
        $*v$:  $M$["m@" + concreteValue($v$)] = abstrEval($e$, $M$);
      return $M$;    /* $M$ is modified */
    assert($cond$):
      return $M$;    /* $M$ is unchanged */
}

procedure abstrEval($e$ : $e$, $M$ : $abstrmap$) returns $abstrval$ {
  $m_1$, $m_2$ : $abstrval$;
  match $e$ with
    $v$: return lookupAndStore($v$, $M$);
    $*v$:  return lookupAndStore("m@" + concreteValue($v$), $M$);
    $c$:  return < c = $c$ , v = nil >;
    $e_1 \circ e_2$:
      $m_1$ = abstrEval($e_1$);  $m_2$ = abstrEval($e_2$);
      if ($m_1$ is nil || $m_2$ is nil)  return nil;  /* NON-LINEAR */
      else if ($m_1$.v is nil && $m_2$.v is nil)
        return < c = $m_1$.c $\circ$ $m_2$.c , v = nil >;
      else if ($\circ \in \{+, -\}$)
        return < c = $m_1$.c $\circ$ $m_2$.c , v = $m_1$.v $\circ$ $m_2$.v >;
      else if ($\circ \in \{*\}$)
        if ($m_1$.v == nil)
          return < c = $m_1$.c $\circ$ $m_2$.c , v = $m_1$.c $\circ$ $m_2$.v >;
        else if ($m_2$.v == nil)
          return < c = $m_1$.c $\circ$ $m_2$.c , v = $m_2$.c $\circ$ $m_1$.v >;
        else return nil;  /* NON-LINEAR */
      else return nil;  /* NON-LINEAR */
}

procedure lookupAndStore($v$ : $v$, $M$ : $abstrmap$) returns $abstrval$ {
  if ($v$ is not a key in $M$)
    $M[v]$ = < c = concreteValue($v$) , v = nil >;
  return $M[v]$;  /* $M$ may be modified */
}
```

Figure 3.4: Second Part of the Pseudocode for Our Symbolic Analysis.

3.3.2 Linking Loops to Input

When the symbolic analysis discussed in Section 3.3.1 is complete, the symbolic expressions for program state variables that our tool has produced will be in terms of the trip count variables. To obtain the relationship between the program state variables and the input, we need to obtain the relationship between the trip count variables and the input. In general, such relationships might be very complicated. However, we leverage the observation that often such trip count variables relate to certain features of the structure of the input such as the length of a variable-length field (such as a string) or the number of records of the same type (called *iterative fields*).

To precisely capture these repetitive features of program inputs, which are missing from descriptions like context-free grammars, we introduce the concept of *auxiliary* attributes. For instance, we introduce *length* attributes to represent the size of fields that might vary in length, and *count* attributes to represent the number of times iterative fields are repeated. Auxiliary attributes are associated with grammatical units at any level (e.g., terminals and non-terminals in a context-free grammar), such as Length(URI) for the length of a URI field in the HTTP grammar. They can also be systematically added to an existing parser as an attribute grammar (as in `yacc` [61]); for instance, the length for a non-terminal in a rule can be computed as the sum of the lengths on the right-hand side of the rule. In some cases, the value of an auxiliary attribute is provided in another field of the input. Our technique can take advantage of auxiliary attributes that appear in the input in this way, but it also uses them in ways that do not require them to appear in the input.

The goal for the linking step is to identify loop-computed values in the program that represent auxiliary attributes; for instance, if a loop is used to compute the length of a field. Previous work [20] shows that automatic inference of variables that iterate over multiple variable-length fields is feasible, and more recently Caballero et al. show how to relate certain program variables to features of an input grammar [17]. We use similar techniques based on the same intuition; we determine that a loop's iteration count is the length of a field if its exit condition checks either a delimiter for the field or a value derived from a length or an auxiliary attribute of the field. In more detail, we use the following steps:

1. *Relate data-dependent bytes to fields.* As in traditional concolic execution, our tool determines for each variable in the trace which input byte(s) (identified by offset) it directly depends on. Our tool also parses the input according to the known grammar, and so determines which protocol field contains each input byte. Therefore, one simple way of matching variables with one or more input fields is to combine these two mappings. For instance, in the example of Figure 3.1, the buffer `URI` contains the contents of the field URI.

2. *Identify variable length fields, counts, and delimiters*. The input grammar also identifies which fields correspond to the lengths or iteration counts of other fields, and our tool maps this information through direct dependencies to determine program variables that represent lengths and counts. Also, we use the grammar to determine which values are used as delimiters to signal the end of a variable-length field. For instance, in the HTTP grammar, the field URI is delimited by a space character.

3. *Identify variables used in loop exit conditions*. By analyzing loops as described in Section 3.5, our tool determines which variables are used in the conditions that determine when to exit a loop. For instance, the loop on lines 26–27 of Figure 3.1 is guarded by a condition on the variables `i` and `uri_len`.

4. *Recognize loops over delimited fields*. If the exit condition of a loop compares bytes of a field to a value that is the delimiter of the field, then we link the iteration count of the loop to the length of the field. For instance, in Figure 3.1, the loop on lines 12-16 compares each byte of the URI field to a space, which is known from the grammar to be the delimiter of the URI, so the execution count of that loop is the length of the field ($TC_1 =$ Length(URI)). In other situations, a loop may process several bytes on each iteration, which gives a relation with a scale factor. For instance, if each iteration processes a 4-byte word, the field length is equal to 4 times the loop trip count.

5. *Recognize loops over counted fields*. If the exit condition of a loop compares a variable to a value that is identified in the grammar as the length of a field or the counter for a repeated field, then we link the iteration count of the loop to that length or count field. As in the case of a delimited field, the scale factor between the field and the trip count may not be 1, for instance if a loop process several items in each iteration.

While these techniques are not enough to recognize every loop that might be written, they represent the most common patterns, and we have found them to be sufficient to capture the relationships for both length and count attributes in practice.

3.4 Applying LECE

In this section we describe how to apply LECE to test generation and in problems about security bugs in software. First, we describe the primitive operation of using LECE to determine how a given predicate might be satisfied during program execution: on a single program path, but perhaps involving different numbers of loop iterations. We then show how to use this primitive for improving coverage in test generation, discovering previously unknown security bugs, and diagnosing the cause of a bug given only an execution that exercises it.

3.4.1 Loop-extended Condition Analysis

A basic use of traditional concolic execution is to determine the conditions under which a predicate at a program location can be true. For instance, the predicate might be a branch condition, a programmer-provided assertion, or an array bounds check. We start with the predicate (which we will call the *query predicate*), associated with a program point, and an execution that reaches that point, but does not satisfy the predicate. Then the task is to determine the conditions on an input to the program that could cause execution to follow the same path, but cause the query predicate to be true. Using loop-extended concolic execution, we enhance this condition analysis by taking into account other program executions that are similar to the observed one, but might involve different numbers of loop executions. Once the predicate has been chosen, this loop-extended condition analysis takes the following 3 steps:

1. *Derive symbolic expressions in terms of inputs.* Given the original execution trace, our tool first performs loop-extended concolic execution on the trace as described in previous sections. The result of this step gives a symbolic expression for each program state variable that depends on the inputs, including both data dependencies and control dependencies introduced by loops.

2. *Instantiate query predicate.* Our tool instantiates the query predicate by using the symbolic expression computed for each variable that appears in the predicate.

3. *Solve constraints.* The query predicate can be satisfied if there exist inputs to the program that simultaneously cause it to reach the location of the predicate, and satisfy the predicate. Therefore, our tool conjoins a path condition with the query predicate, and passes this formula to a decision procedure to determine if it is satisfiable. Constraints in the path condition that arise from loop exit conditions are removed, since they are superseded by loop-dependent symbolic expressions. Our implementation uses STP [48], an SMT solver that represents machine values precisely as bounded bit vectors. If the formula is solvable, STP returns a satisfying assignment to its free variables, which represent particular input bytes and auxiliary attributes.

A grammar-based input generation tool [11, 52] can then be used to produce a version of the initial input, modified according to the satisfying assignment, which is a candidate to satisfy the predicate. When the constraints require that a length or a count be larger, our approach heuristically repeats elements from the initial input until the result is long enough.

3.4.2 Uses for Loop-extended Conditions

Loop-extended condition analysis has many applications. In this section, we describe three: improving the coverage of test generation based on concolic execution, discovering violations of security properties, and diagnosing the exploit conditions of a security flaw.

3.4.2.1 Improving Test Generation

Test generation is the task of discovering inputs to a program that cause it to explore a variety of execution paths. Traditional concolic execution can be used in an iterative search process to find such inputs [22, 53, 102], but it does not cope well with program branches that involve loop-dependent values; using LECE instead allows test generation to achieve higher coverage.

The basic operation in such an iterative search is to take an execution path and a branch along that path, and *reverse* the branch: find an input that causes execution to reach that branch, but then take the opposite direction. Reversing a branch is just an application of the primitive of Section 3.4.1, where the query predicate is a branch condition or its negation. The benefit of using loop-extended concolic execution instead of traditional concolic execution in test generation can be seen in two aspects: First, an LECE-based exploration is able to reverse branches whose conditions involve loop-dependent values; in a tool based on TCE, by contrast, loop-dependent values are not considered symbolic. Second, an iterative search performed with LECE is more directed, since the conditions it reasons about capture the effect on values computed in loops. For instance, if a subsequent branch depends on a loop-derived value, LECE-based search requires only one iteration to determine a number of iterations of the loop to reverse the condition. The length check on line 23 in the example of Figure 3.1 shows this benefit: an LECE-based generation tool can immediately construct an input with a long-enough version field, because the length is a symbolic variable, while a TCE-based tool could only stumble on such an input by trial and error.

3.4.2.2 Vulnerability Discovery

Many classes of security vulnerabilities can occur when a *security predicate* is violated during program execution. For instance, given a program that writes to an array, a buffer overflow occurs if the index of a write to an array is outside of the correct bounds. In a program that uses machine integers to compute the length of a data structure, an integer overflow vulnerability occurs if a computation gives the wrong result when truncated to word size. To check whether program logic is sufficient to prevent such failures, the problem of vulnerability discovery, or "fuzzing," asks whether there is a program input that could violate the security predicate. Vulnerability discovery is similar to test case generation; the only difference is the additional checking of a security predicate at each dangerous operation. Thus, like test generation, it can be performed using our loop-extended condition analysis: the query predicate is just the negation of the security predicate.

Loop-extended concolic execution is a particularly good match for discovering vulnerabilities related to input processing, because the data structure size values that are misused in buffer overflow and integer overflow vulnerabilities are often processed using loops. The buffer overflow in Figure 3.1 is typical in this way. Depending on the security property, some preprocessing might be needed to precisely define the security predicate describing how an operation might be unsafe: for instance, when checking for a buffer overflow, to determine the length of the vulnerable buffer. We will discuss some practical aspects of such preprocessing in Section 3.5.

3.4.2.3 Vulnerability Diagnosis

If a vulnerability has already been exploited by an attacker, another important application is diagnosing it: extracting a set of *vulnerability conditions* (general constraints on the values of inputs that exploit the vulnerability). Diagnosis is an important problem in security because vulnerability conditions are useful for automatically generating signatures to search for or filter attacks, or to help a security analyst understand a vulnerability.

Vulnerability diagnosis is again based on the loop-extended condition analysis primitive of Section 3.4.1: in fact, the combination of a path predicate and a negated security predicate gives a vulnerability condition. However, concolic execution typically generates thousands of constraints, so our tool performs several optimizations to simplify them into a smaller set, as discussed in Section 3.5. Such simplification is particularly important for applications involving manual analysis, but a compact condition is also more efficient for use by later automated tools.

Some forms of vulnerability diagnosis could be performed using TCE, but a TCE-based diagnosis would be too narrow for many applications, including most buffer overflows. For instance, a TCE-based diagnosis of the web server in Figure 3.1 could capture some generality in the contents of the input fields, but it would restrict their lengths to the particular values seen in the sample exploit. A filter based on such a diagnosis could be easily bypassed by an attack that used a different length URI. By contrast, LECE finds conditions that are more general; for instance, in the example of Figure 3.1, it finds that `msgbuf` can be overflowed by inputs of arbitrary size, as long as the sum of the lengths of two fields is at least 99.

3.5 Implementation

We have implemented the core loop-extended concolic execution component described earlier in OCaml, and the protocol format linkage in OCaml combined with C and Python code to integrate with off-the-shelf parsers. We utilized our existing binary analysis infrastructure [10, 105] for taking an execution trace and getting the semantics of x86 instructions.

The rest of this section describes several additional components we developed to realize our proposed primitives and heuristics that make this approach practical when working with binaries.

Memory layout extraction. Our tool infers on its own the memory allocations made by the program at different points during the program execution. Then, the tool checks these allocations for overflows in pointer accesses. When dealing with dynamic allocation, it uses the arguments to memory allocation functions as being recorded by TEMU (part of our infrastructure discussed in Section 2.4). For stack-based memory accesses, we implemented a stack bound inference technique called stack analysis [100], though more detailed techniques [4, 5] could alternatively be used.

Loop information extraction. Our infrastructure uses the IDA Pro tool [60] to disassemble binaries and we reused the static loop detection algorithms [84, pp. 191–197] existing in our infrastructure [17]. There are two notable additional caveats which were useful for obtaining results for our case studies.

1. *Addition of dynamic edges.* The presence of indirect call and jump instructions hinders static CFG extraction: an analysis may completely miss code blocks that are reachable only through indirect jumps. Our static control flow graph extraction is

supplemented with indirect jump targets observed in the trace, which allow many more loops to be discovered. For instance, such loops were critical to obtaining accurate results in the SQL Server case study of Section 3.6.2.

2. *Irreducible loops.* Unlike in high-level languages, loops in binaries are often irreducible. We dealt with this by employing node-splitting techniques [1, pp. 684–685] to make loops reducible so that they can be identified in the control flow graph by means of searching for back edges.

Protocol Grammar. Our existing infrastructure interfaces with Wireshark [116], an off-the-shelf IDS/IPS, to obtain protocol grammars of network protocols we study.

Input Generation. We find that a relatively simple input generation approach works well with our LECE implementation: when a constraint requires that a length or count be larger, we repeat elements from the initial input until the result is long enough. In more general examples where the field being extended is subject to more additional constraints, one could also leverage grammar-based input generation approach [11, 52].

Constraint simplification. Our tool performs live-variable analysis to remove irrelevant constraints. It then performs constant folding on the remaining constraints, and simplifies them using the algebraic simplification routines built-in with the STP constraint solver [48].

3.6 Experimental Evaluation

We evaluated the effectiveness of loop-extended concolic execution by applying it to discovery and subsequent diagnosis of buffer overflow vulnerabilities. We selected two kinds of subject programs for this evaluation. For comparison with other implementations, which require source code and/or run only on Linux, we use standard benchmark suites containing known overflows. To test the practical utility of our tool, we use real-world Windows and Linux applications with historic vulnerabilities. Our tool discovers all the benchmark overflows, as well as those in real-world applications, by generating just a few candidate inputs.

In addition to a subject binary program and an initial program input, our system also requires the following data to complete each of the experiments.

- Extra inputs
  - Grammar of the program input (obtained using Wireshark)
  - Program disassembly (required for loop detection, obtained using IDA Pro)
- Data automatically inferred by our system
  - Stack and heap memory layout (required for overflows detection)
  - Locations of indirect jumps/calls (to improve loop detection)

3.6.1 Evaluation on a Benchmark Suite

As benchmarks, we used a set of 14 samples extracted from vulnerabilities in open-source network servers (BIND, Sendmail, and WuFTP) by researchers at the MIT Lincoln Laboratories [122], which range between 200 and 800 lines of code each. (These are the same benchmark programs used by Xu et al. [120].)

Replacing TCE with LECE would be beneficial throughout input space exploration in vulnerability discovery, since symbolic expressions for loop-dependent values allow more branches to be reversed, as discussed in Section 3.4.2.1. However, it can be difficult to fairly compare concolic execution tools on an end-to-end basis, because of differences in input assumptions and search heuristics. Therefore, we confine our evaluation to the last stage of vulnerability search by starting both our tool and a TCE tool with a program input that reaches the line of code where a vulnerability occurs, but does not exploit it. These inputs are short and/or close to usual program inputs, so they could be found relatively easily by either a TCE-based or an LECE-based approach (though the time required would still be highly dependent on the initial input and search heuristics used). Therefore, the results on these inputs provide a bound on the performance of an end-to-end system: if a tool is unable to find a vulnerability given the hint of a nearby input, it would also be unable to find it starting from a completely unrelated input.

Results and New Bugs. The upper half of Table 3.2 shows the results of our tool on the Lincoln Labs overflow benchmarks. The first column identifies each benchmark, and the second column summarizes the input grammar our tool uses. The third and fourth columns show the initial input our tool started with, and the exploit input it found. The fifth column shows the number of candidate inputs our tool generates (after the slash), and the number of those that in fact cause an overflow (before the slash). The sixth column shows the total runtime of our tool, starting with the initial input trace and including all the discovered

Program	Input Format	Initial Input	Exploit Input	Bug / Candidate	Time (s)	Loop-Dep. Conditions
BIND 1	DNS QUERY	104 bytes, RDLen=48	RDLen=16	1/5	2511	16
BIND 2	DNS QUERY	114 bytes, RDLen=46	RDLen=30	1/4	2155	12
BIND 3	DNS IQUERY	39 bytes, RDLen=4	RDLen=516	1/2	586	13
BIND 4	DOMAINNAME	“web.foo.mit.edu”	“web.foo.mit.edu” (64 times)	1/1	4464	52
Sendmail 1	Byte Array	“<><><>”	“<>” (89 times)	4/5	672	1
Sendmail 2	struct passwd (Linux)	(“”,“root”,0,0,“root”,“”,“”)	(“”,“root”,0,0,“rootroo”,“”,“”)	1/1	526	38
Sendmail 3	[String]^N	[“a=\n”]^2	[“a=\n”]^{59}	1/4	626	18
Sendmail 4	Byte Array	“aaa”	“a” (69 times)	1/1	633	2
Sendmail 5	Byte Array	“\\\”	“\” (148 times)	3/3	18080	6
Sendmail 6	OPTION○’ ’○ARG	“-d aaaaaaaaaa-2”	“-d 4222222222-2”	1/1	676	11
Sendmail 7	DNS Response Fmt	TXT Record : “aaa”	Record : “a” (32 times)	1/1	237	16
WuFTP 1	String	“aaa”	“a” (9 times)	2/2	483	5
WuFTP 2	PATH	“aaa”	“a” (10 times)	1/1	197	29
WuFTP 3	PATH	“aaa”	“a” (47 times)	1/1	109	7
GHttpd	Method○URI○Version	“GET /index.html HTTP/1.1”	“GET ”+188 bytes + “ HTTP/1.1”	2/2	1562	41
SQL Server	Command○DBName	x04 x61 x61 x61	x04 x61(194 bytes)	1/3	205	1
GDI	(Not required)	1014 bytes, INP[19:18]=0x0182	INP[19:18]=0x4003	1/1	353	2

Table 3.2: Discovery Results for Benchmarks and Real-world Programs. A circle (○) represents concatenation. In $[X]^k$, k denotes the auxiliary count attribute specifying the number of times element X repeats.

overflows. (The seventh column will be discussed in Section 3.6.3.) All experiments were performed on a 3GHz Intel Core 2 Duo with 4GB of RAM.

Our LECE tool discovers most of the bugs in just a few minutes, requiring only a few candidate inputs each. In each case, we supplied a small benign input, and the tool automatically found that a longer input could cause an overflow. Our tool also discovered an apparently new bug in one of the Lincoln Labs benchmarks: in addition to the known overflows (marked with `/* BAD */` comments in the benchmark code) our tool finds a new overflow on line 340 of the function `parse_dns_reply` in Sendmail benchmark 7. (In the other cases where our tool reports multiple overflowing inputs, they were a set of related errors marked in the benchmark.)

Comparison with Splat. Xu et al. [120] suggest a different approach to making TCE work better for certain buffer overflows, by abstracting over the length of string buffers. Specifically, their length abstraction technique requires programmer-supplied source code annotation to mark a chosen prefix of the relevant buffer's contents as symbolic. In contrast, our technique automatically extracts memory buffers and their dependency on the input fields using a combination of static and dynamic analysis. More importantly, LECE does not need any information about string-manipulating functions, but instead uses input grammar to assist its analysis. Our key enhancement to handle loop dependencies is practically sufficient to reason about the implementations of the string functions for our applications. As a result, LECE can reason about vulnerabilities present in custom operations on array inputs that may not use any common string operations (examples of these are available in our studied benchmarks).

Though the Lincoln Labs benchmarks were also studied by Xu et al. [120], a head-to-head empirical comparison was not possible. Unfortunately, because of the way the original benchmarks are designed to be self-contained, it was unclear which buffers (and which parts) were annotated as program inputs in their work. For instance, the BIND 2 benchmark exercises code from BIND that parses a DNS packet, and also includes code to generate an appropriate packet. In Xu et al. [120], it was unclear which value in the packet generation process was treated as the input. As shown in Table 3.2, we considered the whole packet itself to be the input, so that only an input that is a mostly syntactically correct packet will cause an overflow. We believe our choice makes for a more realistic evaluation, but it implies that a direct comparison of the tools' execution times would not be meaningful.

Our tool was able to find exploits for the two benchmarks (Sendmail 1 and 5) on which Splat times out. (In the case of Sendmail 5, the total running time of our tool to evaluate 3 candidate inputs is longer than the two-hour timeout used with Splat, but our tool reports its first vulnerability before two hours have elapsed.) On the remaining benchmarks, our tool reproduces Splat's positive results on the complete programs.

Accuracy of candidate inputs. In the fifth column, Table 3.2 shows the number of candidate test inputs our tool generated in the process of finding each exploit. The fact that only a few tests were required (on average, 62.5% of the candidates our tool generates are real exploits) demonstrates the targeted nature of LECE-based search: the tool efficiently chooses appropriate loop iteration counts and prunes buffer operations that are safe, concentrating on the most likely vulnerability candidates. Of course, since the candidates are concrete inputs that can be automatically tested, failed candidates are not reported: the tool gives no false positive results.

3.6.2 Evaluation on Real-World Programs

As full-scale case studies, we took 3 real-world Windows and Linux programs which are known to have buffer overflow vulnerabilities. These include the program targeted by the infamous Slammer worm in 2003 [83], the one affected by a recent GDI vulnerability in 2007 [78], and an HTTP server [50]. Table 3.2 summarizes the vulnerabilities in these programs and the input grammars our tool used. We gave benign initial inputs to these programs that are representative of normal inputs that they would receive in practice.

Starting with a benign input, our tool uses just one iteration of LECE to discover buffer overflows in all 3 real world programs. The bugs found in the GDI and SQL cases are the same reported earlier in these programs, as we manually confirmed. For ghttpd, our tool discovers two buffer overflow vulnerabilities in the `Log` function in `util.c`. One of these is described in previous research using this subject program [31]. The new overflow involves a separate buffer and would need a separate fix. These results are shown in Table 3.2; next we explain each vulnerability in more detail.

GHttpd vulnerability. GHttpd is a Linux web server; we use version 1.4.3. We send an initial benign input, `GET /index.html HTTP/1.1`, to the running web service, and it responds normally. Given a trace of this execution and the HTTP grammar, our tool discovers 2 potential buffers to overflow and generates candidate exploits for each. These inputs are the same as the initial input except that their URI fields have lengths of 188 and 140 bytes respectively. Testing confirms that both candidates indeed cause overflows: the shorter request overflows one buffer, and the longer one overflows both that buffer and a subsequent one.

SQL Server vulnerability. This vulnerability is a stack-based overflow in Microsoft's SQL Server Resolution Service (SSRS), which listens for UDP requests on port 1434. Based on its specification [79], one valid message format contains 2 fields: a header byte of value 4, followed by a string giving a database name. We send the SSRS service a benign request that consists of the header byte and a string "`aaa`", to which the service responds correctly. Given the trace and the input grammar, our tool finds 3 potential buffers to overflow and generates one candidate inputs for each. Our automated testing reports that one candidate, which is 195 bytes long, overflows a buffer that is the same one exploited by the SQL Slammer worm. (The other two candidate inputs are longer than the maximum-length UDP packet, so they are discarded during testing and not reported.) The fact that such large inputs could be generated in a single step, rather than via a long iteration process, shows the power of LECE.

GDI vulnerability. This vulnerability in the Microsoft Windows Graphic Rendering Engine was patched in 2007. We created a benign and properly formatted WMF image file using Microsoft PowerPoint, containing only the text "`aa`"; the file is 1014 bytes long. We attempt to open the file using a sample application and record the program execution. Without using an input grammar, our tool discovers a potential buffer read overflow and creates an exploit input, which crashes the sample application. The only differences between the exploit and the benign input are the values in bytes 18 and 19 (shown in Table 3.2). Comparing with a grammar for the WMF format, these bytes correspond to the size of the image field.

3.6.3 Further Applications

Improving test coverage. Though our evaluation does not focus on the exploration phase of vulnerability detection, our experiments do demonstrate a feature of loop-extended concolic execution that makes it more effective in obtaining input space coverage. As described in Section 3.4.2.1, LECE improves on TCE by finding symbolic expressions for more branch conditions that depend on the number of times loops execute, making it possible for a coverage tool to reverse them. To measure this effect, we give in the last column of Table 3.2 the number of branches for which our tool found a loop-dependent condition but no directly input-dependent condition, so that an LECE-based tool would be able to reverse them but a TCE-based tool would not. The count is a number of unique program-counter locations (i.e., static and context-insensitive), and excludes loop exit conditions. For instance, one of the 29 loop-dependent conditions in WuFTP 2 is a length check (on line 464) intended to prevent the buffer overflow. Because the check is faulty, it is false on both our benign and exploit inputs, but exploring both sides would be critical for an exploration task, such as verifying the lack of overflows in a fixed version. The condition is immediately apparent to our tool, but would not be considered symbolic under standard TCE.

Vulnerability diagnosis. Our tool can also be used for vulnerability diagnosis: to find a general set of conditions under which an exploit occurs. Diagnosis is most useful when a vulnerability is already being used by attackers, and it is important to understand and defend against attacks quickly: vulnerability conditions can accelerate or replace manual analysis of an exploit, and be used to generate filters to detect or block attacks. But to be useful, such conditions must be broad enough to cover a large class of attacks.

We used our tool to perform diagnosis on the same real-world programs described in Section 3.6.2. Either a publicly available exploit, or the exploits generated by our discovery tool, could be used and produce the same results.

Our tool's diagnoses, summarized in Table 3.3, are more accurate and usable than those given in previous work [37]. For instance, for the Microsoft SQL Server vulnerability, the condition our tool generates states that the vulnerable field's length must be greater than 64 bytes, whereas the buffer overrun vulnerability condition generated in previous work states that the length must be at least 97 bytes [37]. This difference turns out to be significant. Because we have no access to source code, we validated our results experimentally by supplying inputs of various sizes to the server. We found that when the vulnerable field has a size larger than 64 bytes, the overflow overwrites pointers with invalid values, causing an exception when these values are dereferenced.

Also note that most diagnoses of buffer overflows, including the GHttpd and SQL Server examples shown in Table 3.3, could not be produced by a standard TCE tool, which lacks even a notation to refer to the length of an input field.

3.7 Limitations

Although we found that LECE provides enhancement to concolic execution and can be applied to various security-related applications as discussed in Section 3.6, it is valuable to discuss some key limitations of the technique.

First, LECE uses static loop detection technique instead of detecting them during program execution, which would make the overall technique dynamic and applicable for any

Program	Buffer size (bytes)	Condition for overflow	Constraint generation time (s)
GHttpd (1)	220	`URI.len > 172`	420 + 23
GHttpd (2)	208	`URI.len > 133`	420 + 140
SQL Server	128	`DBName.len > 64`	192
GDI	4096	`(2·INP[19:18])>>2 < 0`	200

Table 3.3: Diagnosis Results on Real-world Software. Generation time for GHttpd consists of the pre-processing time (420 s) and the post-processing time (23 s and 140 s) for each condition.

executables. As a result, LECE requires a control flow graph of a program under analysis which may be difficult to retrieve for obfuscated programs. We choose the static approach because it provides the loop exit conditions which are crucial to our overall approach. The dynamic approach, on the other hand, would require unproven heuristics to identify loop exit condition. Second, it relies heavily on the presence of input grammar, which may not be readily available. Even though previous work [20] has proposed a technique for automatic inference of proprietary input formats, the correctness of the resulting formats cannot be fully guaranteed and the technique is not robust against obfuscation. Third, LECE requires a heuristic for extending the length of an input field, by repeatedly concatenating elements from the initial input field until it reaches the desired length. Although the heuristic works fine in our experiment, it is possible that the extended field may be subject to additional constraints. Finally, LECE only consider linear relationship between loop trip counts and variables, which is not applicable to many input formats that contain nested fields.

3.8 Related Work

This section discusses two classes of related research: first, other work on analysis approaches similar to our loop-extended concolic execution; then, work that also addresses the problem of discovering and/or diagnosing buffer-overflow attacks.

3.8.1 Analysis Approaches

Extensions to traditional concolic execution. Several previous approaches have extended traditional concolic execution with additional information about the program or its possible inputs. Previous grammar-based approaches [17, 51, 52, 74] have taken advantage of knowledge of which program inputs are legal to reduce the size of the search space when generating new inputs. By comparison, our use of an input grammar in Section 3.3.2 is focused on extracting more information from a single execution. The Splat tool of Xu et al. [120] also targets the problem of buffer-overflow diagnosis, but they do not explicitly model loop constructs as in loop-extended concolic execution. An empirical and analytical comparison to their approach is presented in Section 3.6.1. Pre- and post-conditions can summarize the behavior of a function so that it need not be reanalyzed [51], similar to how our approach avoids the need to reanalyze with different numbers of loop iterations. If repeated constraints are generated, they can also be later removed by optimizations such as constraint subsumption [54].

Static analysis. Determining linear (technically, "affine") relationships among the values of variables, as our analysis in Section 3.3.1 does, is a classic problem of static program analysis, pioneered by Karr [62]. Like many properties that involve multiple variables, it can potentially become expensive. For instance the polyhedron technique [33] requires costly conversion operations on a multi-dimensional abstract representation. More recent research has considered restricted abstract domains that allow for more efficient computation, such as "octagons" [81] and "pentagons" [72]. The techniques of Müller-Olm and Seidl [85] have the advantage of giving precise results even with respect to overflow, but their runtime is a high power of the number of variables in a program (k^7 for the interprocedural case). Random analysis [56] can also be used to determine linear relationships, with a small probability of error. For the simpler case we consider, it is sufficient to take a more efficient non-relational approach: we express the values of program variables not in terms of each other but in terms of a small set of auxiliary trip-count variables.

3.8.2 Discovering and Diagnosing Buffer Overflows

Buffer-overflow vulnerabilities are a critical security challenge, and many approaches target them. Sound static analysis holds the possibility of eliminating false negatives, but in practice buffer overflow checking is difficult enough that sound analysis is possible only for small programs with extensive user annotation [44]. More comparable to our approach are scalable bug-finding tools [47, 119]. However, pure static analysis approaches suffer from false positives, which tool users must examine by hand. For instance, one comparison [122] using the same benchmarks we use in Section 3.6.1 found that many tools produced so many false positives they did only slightly better than chance. Dynamic analysis techniques, on the other hand, avoid false positives by examining programs as they execute [32, 34, 88]. However, the requirement of running on all executions means that the overhead of dynamic analysis tools can limit their applicability. Concolic execution combines static and dynamic techniques to generalize from observed executions to similar unobserved ones, and loop-extended concolic execution extends this generalization to include loops.

Our vulnerability diagnosis using loop-extended concolic execution extends previous diagnosis approaches based on traditional concolic execution [13, 14, 31]. Bouncer [31] employs source-code-based static alias analysis along with TCE. ShieldGen [37] uses a protocol-specification-based exploration of the input space to diagnose a precise vulnerability condition. However, in contrast to our work, it treats the program as a black-box, ignoring the implementation. In addition, it does not capture complex relationships between fields that may be necessary to exploit a vulnerability. For instance, as its authors point out, ShieldGen cannot capture the condition that the combined length of two fields must exceed a buffer size for exploit (as in the example of Section 3.2), while our techniques can.

3.9 Conclusion

We propose loop-extended concolic execution, a new type of concolic execution that gains power by modeling the effects of loops. It introduces trip count variables with a symbolic analysis of linear loop dependencies, and links them to features in a known input grammar. We apply this approach to the problem of detecting and diagnosing buffer overflow vulnerabilities, in a tool that operates on unmodified Windows and Linux binaries. Rather than trying a large number of inputs in an undirected way, our approach often discovers an overflow on the first candidate it tries. Our tool finds all the vulnerabilities in the Lincoln Labs benchmark suite and gives accurate symbolic conditions describing real vulnerabilities. These results suggest that loop-extended concolic execution has the potential to make many kinds of program analysis, including important security applications, faster and more effective.

Chapter 4

Decomposition and Re-stitching

4.1 Introduction

Vulnerability discovery in *benign* programs has been an important task in software security: identifying software bugs that may be remotely exploitable and creating program inputs that demonstrate their existence. However, finding vulnerabilities in *malware* has not been studied. Do malicious programs have vulnerabilities? Do different binaries of the same malware family share vulnerabilities? How do we automatically discover vulnerabilities in malware? What are the implications of vulnerability discovery to malware defense, law enforcement and cyberwarfare? In this thesis we take the first step in addressing these questions. In particular, we propose new symbolic reasoning techniques for automatic input generation in the presence of complex *encoding functions* and demonstrate the effectiveness of our techniques by examining and finding bugs in real-world malware. Our study also shows that vulnerabilities can persist for years across malware revisions. We hope our work will spur discussions in the implications of malware vulnerability discovery to malware defense, law enforcement and cyberwarfare.

Concolic execution and related techniques can be used for vulnerability discovery in malware just like in benign software. However traditional concolic execution is ineffective in the presence of certain common computation tasks, including the decryption and decompression of data, and the computation of checksums and hash functions. We call these *encoding functions*. Encoding functions result in symbolic formulas that can be difficult to solve, which is not surprising, given for instance that cryptographic hash functions are designed to be impractical to invert [86][1]. Encoding functions are used widely in mal-

[1]Inversion of hash and checksum function in this thesis refers to finding any input that would provide the

ware (and also in many benign applications). In our experiments, the traditional concolic execution approach fails to explore the execution space of the malware samples effectively.

To address the challenges posed by the presence of encoding functions, we propose a new approach, *stitched* concolic execution. This approach first identifies potential encoding functions and their inverses, if any. Then, it decomposes the symbolic constraints from the execution, separating the constraints generated by each encoding function from the constraints in the rest of the execution. The solver does *not* attempt to solve the (hard) constraints induced by the encoding functions. It focuses on solving the (easier) constraints from the remainder of the execution. Finally, it re-stitches the solver's output by either using the inverses of the encoding functions or by solving related constraints, creating a program input that can be fed back to the unmodified program.

For instance, our approach can automatically identify that a particular function in an execution is performing some expensive computation on the input, e.g., decrypting the input. Rather than using symbolic execution inside the decryption function, it applies symbolic execution on the outputs of the decryption function, producing constraints for the execution after the decryption. Solving those constraints generates an unencrypted message. Then, it executes the inverse (encrypt) function on the unencrypted message, generating an encrypted message that can be fed back to the program.

More generally, we identify two kinds of computation that make such decomposition possible: computations that transform data into a new form that replaces the old data (such as decompression and decryption), and side computations that relate a constrained value to an otherwise unconstrained value (such as checksums). For clarity, we explain these techniques in the context of concolic execution, but they are equally applicable to concrete fuzz (random) testing (e.g., [45, 106]) and taint-directed fuzzing [49].

We implement our approach as a set of additions to a system for automated concolic execution of off-the-shelf x86 executables in binary form, implemented using our BitBlaze infrastructure [10, 105]. Our re-stitching approach enables the first automated study of bugs in malware: our tool finds several new, remotely trigger-able bugs in prevalent malware programs such as botnet clients and trojans. Vulnerabilities in botnet clients could allow a third party to terminate or take control a bot, so they are a powerful tool for either defensive or malicious purposes. To confirm the value of our approach, we show that our tool would be unable to find most of the bugs we report without the new techniques we introduce.

Malware vulnerabilities have a great potential for different applications such as malware removal or cyberwarfare. Some malware programs such as botnet clients are de-

same hash as the original input.

ployed at a scale that rivals popular benign applications. For instance, the recently-disabled Mariposa botnet was sending messages from more than 12 million unique IP addresses at the point it was taken down, and stole data from more than 800,000 users [69]. Our goal in this research is to demonstrate that finding vulnerabilities in widely-deployed malware such as botnet clients is technically feasible. However, the implications of the usage of malware vulnerabilities require more investigation. For example, some of the potential applications of malware vulnerabilities raise ethical and legal concerns that need to be addressed by the community. Thus, another goal of this research is to raise awareness and spur discussion in the community about the positives and negatives of the different uses of malware vulnerabilities.

4.2 Problem Definition & Overview

In this section, we describe the problem we address and give an overview of our approach.

4.2.1 Problem Definition

Our problem is how to perform concolic execution in the presence of encoding functions.

Often there are parts of a program that are not amenable to concolic execution. A class of common culprits, which we call *encoding functions*, includes many instances of decryption, decompression, and checksums. For instance, consider the code in Figure 4.1, which is an idealized example modeled after a botnet client. After receiving a message (`struct msg`) from the network, it first decrypts the body of the message using AES [39], verifies that it has a correct SHA-1 hash [86], and then takes a malicious action such as sending spam based on a command in the message. Concolic execution attempts to create a new valid input by solving a formula corresponding to the path condition for an execution path. Suppose we run the program on a message that causes the bot to participate in a DDOS attack: at a high level, the path condition takes the form

$$m' = \text{Dec}(m) \wedge h_1 = \text{SHA1}(m') \wedge m'[0] = 101 \tag{4.1}$$

where m and h_1 represent two relevant parts of the program input treated as symbolic: m is the message body `m->message`, and h_1 is the message checksum `m->hash`. Dec represents the operation of AES decryption, while SHA1 is the SHA-1 hash function. To see whether it can create a message to cause a different action, concolic execution will attempt to solve the modified path condition

$$m' = \text{Dec}(m) \wedge h_1 = \text{SHA1}(m') \wedge m'[0] \neq 101 \tag{4.2}$$

```
1  struct msg {
2      long msg_len;
3      unsigned char hash[20];
4      unsigned char message[];
5  };
6  void process(unsigned char* network_data) {
7      int *p;
8      struct msg *m = (struct msg *) network_data;
9      aes_cbc_decrypt(m->message, m->msg_len, key);
10     p = compute_sha1(m->message, m->msg_len);
11     if (memcmp(p, m->hash, 20))
12         exit(1);
13     else {
14         int cmd = m->message[0];
15         if (cmd == 101)
16             ddos_attack(m);
17         else if (cmd == 142)
18             send_spam(m);
19           /* ... */
20     }
21 }
```

Figure 4.1: A Simplified Example of a Program that Uses Layered Input Processing. The encoding functions include decryption (line 9) and a secure hash function for integrity verification (lines 10-12).

which differs from the original in inverting the last condition.

However solvers tend to have a very hard time with conditions such as this one. As seen by the solver, the Dec and SHA1 functions are expanded into a complex combination of constraints that mix together the influence of many input values and are hard to reason about [41]. The solver cannot easily recognize the high-level structure of the computation, such as that the internals of the Dec and SHA1 functions are independent of the parsing condition $m'[0] \neq 101$. Such encoding functions are also just as serious an obstacle for related techniques like concrete and taint-directed fuzzing. Thus, the problem we address is how to perform input generation (such as via concolic execution) for programs that use encoding functions.

4.2.2 Approach Overview

We propose an approach of *stitched* concolic execution to perform input generation in the presence of encoding functions. We first discuss the intuition behind it, outline the steps involved, and then explain how it applies to malware vulnerability finding.

Intuition. The insight behind our approach is that it is possible to avoid the problems caused by encoding functions, by identifying and bypassing them to concentrate on the rest of the program, and re-stitching inputs using concrete execution. For instance in the path condition of formula 4.2, the first and second constraints come from encoding functions. Our approach can verify that they are independent from each other and the message parser (exemplified by the constraint $m'[0] \neq 101$) within the high-level structure of input processing and checking. Thus these constraints can be decomposed, and the solver can concentrate on the remainder. Solving the remaining constraints gives a partial input in the form of a value for m', and our system can then re-stitch this into a complete program input by concretely executing the encoding functions or their inverses, specifically h_1 as SHA1(m') and m as Dec$^{-1}(m')$.

Many encoding functions use keys or seeds which can be observed during program execution. Our approach heuristically assumes that the inverses are using the same key. Thus, it will not work on functions that use public-key cryptography. We discuss this limitation in Section 4.6.2.

Stitched concolic execution. In outline, our approach proceeds as follows. As a first phase, our approach identifies encoding functions (such as decryption and checksums) based on a program execution. Then in the second phase, our approach augments exploration based on concolic execution by adding decomposition and re-stitching. On each iteration of exploration, we decompose the generated constraints to separate those related to encoding functions, and pass the constraints unrelated to encoding functions to a solver. The constraint solution represents a partial input; the approach then re-stitches it, with concrete execution of encoding functions and their inverses, into a complete input used for a future iteration of exploration. If as in Figure 4.1 there are multiple layers of encoding functions, the approach decomposes each layer in turn, and then reverses the layers in re-stitching. Since our approach is used with and based on dynamic analysis, it is not a requirement that it be sound over all possible executions; we judge the re-stitching by the program inputs it generates, which we can check by executing them.

Finding vulnerabilities in malware. We implement the approach as a scaling tool for BitFuzz (discussed in Section 2.4). We use BitFuzz to find vulnerabilities in malware programs. Many such malware samples, e.g., bots, act as network clients that start connections to remote C&C servers. Thus, the input that BitFuzz needs to feed to the program in each iteration is often the response to some request sent by the program. Previous exploration tools for binaries do not support such network clients and have focused on programs that read input from the file system or network servers that receive network data directly from the exploration tool. One goal for BitFuzz is to be able to explore such network client programs without a real connection to the Internet, what we call *Internet in your workstation*. For malware, this means that we do not need to worry about malicious behavior leaking to the Internet. To enable such exploration we have developed a set of tools, which we detail in Section 4.4.

4.3 Stitched Concolic Execution

In this section we describe key aspects of our approach: the conditions under which a program's constraints can be decomposed and re-stitched (Section 4.3.1), techniques for choosing what components' constraints to decompose (Section 4.3.2), and how to repeat the process when there are multiple encoding layers. (Section 4.3.3). An overview of the system architecture is shown in Figure 4.3.

4.3.1 Decomposition and Re-Stitching

In this section we describe the principles of our decomposition and re-stitching approach at two levels: first at the level of constraints between program values, and then more

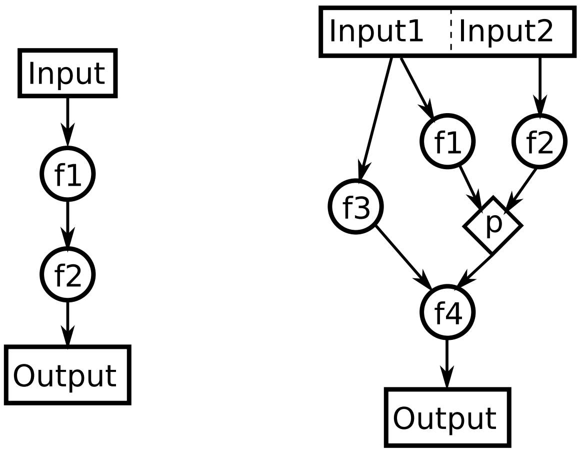


Figure 4.2: A Graphical Representation of the Two Styles of Decomposition Used in Our Approach. Ovals and diamonds represent computations, and edges represent the dependencies (data-flow constraints) between them. On the left is serial layering, while on the right is side-condition layering.

abstractly by considering a program as a collection of functional elements.

4.3.1.1 Decomposing Constraints

One perspective on decomposition is to consider a program's execution as inducing constraints among program values. These are the same constraints that are represented by formulas in symbolic execution: for instance, that one value is equal to the sum of two other values. The constraints that arise from a single program execution have the structure of a directed acyclic graph whose sources represent inputs and whose sinks represent outputs; we call this the *constraint graph*. The feasible input-output pairs for a given execution path correspond to the values that satisfy such a constraint system, so input generation can be viewed as a kind of constraint satisfaction problem.

In this constraint-satisfaction perspective, analyzing part of a program separately corresponds to cutting the constraints that link its inputs to the rest of the execution. For a formula generated by concolic execution, we can make part of a formula independent by renaming the variables it refers to. Following this approach, it is not necessary to extract a component as if it were a separate program. Our tool can simply perform concolic execution on the entire program, and achieve a separation between components by using different variable names in some of the extracted constraints.

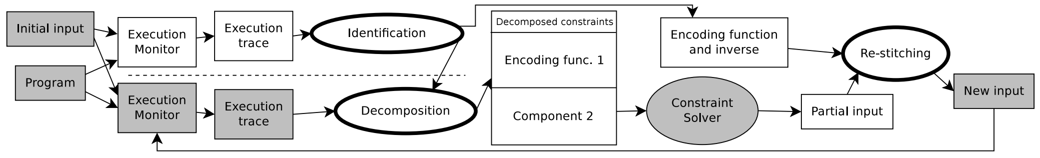


Figure 4.3: Architectural Overview Showing the Parts of Our Decomposition-based Input Generation System. The steps labeled decomposition and re-stitching are discussed in Section 4.3.1, while identification is discussed in Section 4.3.2. The parts of the system shown with a gray background are the same as would be used in a non-stitching concolic execution system. The steps above the dotted line are performed once as a setup phase, while the rest of the process is repeated for each iteration of exploration.

We propose two generic forms of decomposition, which are illustrated graphically in Figure 4.2. For each form of decomposition, we explain which parts of the program are identified for decomposition, and describe what local and global dependency conditions are necessary for the decomposition to be correct.

One set of global dependency conditions are inherent in the graph structure shown in Figure 4.2. If each node represents the constraints generated from one component, then for the decomposition to be correct, there must not be any constraints between values that do not correspond to edges in Figure 4.2. For instance the component f_2 in serial decomposition must not access the input directly.

Serial decomposition. The first style of decomposition our approach performs is between successive operations on the same information, in which the first layer is a transformation producing input to the second layer. More precisely, it involves what we call a *surjective transformation*. There are two conditions that define a surjective transformation. First, once a value has been transformed, the pre-transformed form of the input is never used again. Second, the transformation must be an onto function: every element in its codomain can be produced with some input. A function $y = x^2$ which returns a signed 32-bit integer is an example of functions that do not satisfy this condition. The codomain of this function contains 2^{32} elements, including negative integers which are not possible output of the function. In Figure 4.2, f_1 is the component that must implement a surjective transformation. Some examples of surjective transformations include decompression and decryption. The key insight of the decomposition is that we can analyze the part of the program downstream from the transformation independently, and then simply invert the transformation to re-stitch inputs. For instance, in the example of Figure 4.1, the decryption operation is a surjective transformation that induces the constraint $m' = \text{Dec}(m)$. To analyze the rest of the program without this encoding function, we can just rename the other uses of m' to a new variable (say m'') that is otherwise unconstrained, and analyze the program as if m'' were the input. Bypassing the decryption in this way gives

$$h_1 = \text{SHA1}(m'') \wedge m''[0] = 101 \tag{4.3}$$

as the remaining path condition.

Side-condition decomposition. The second style of decomposition our approach performs separates two components that operate on the same data, but can still be considered mostly independent. Intuitively, a *free side-condition* is a constraint on part of a program's input that can effectively be ignored during analysis of the rest of a program, because it can always be satisfied by choosing values for another part of the input. We can be free to change this other part of the input if it does not participate in any constraints other than those from the side-condition. More precisely, a program exhibiting a free side-condition takes the form shown in the right-hand side of Figure 4.2. The side-condition is the constraint that the predicate p must hold between the outputs of f_1 and f_2. The side-condition is free because whatever value the first half of the input takes, p can be satisfied by making an appropriate choice for the second half of the input. An example of a free side-condition is that the checksum computed over a program's input (f_1) must equal (p) the checksum parsed from a message header (f_2). Section 4.3.2.1 discusses how a free side-condition can be identified.

To perform decomposition given a free side-condition, we simply replace the side-condition with a value that is always true. For instance the SHA-1 hash of Figure 4.1 participates in a free side-condition $h_1 = \text{SHA1}(m'')$ (assuming we have already removed the decryption function as mentioned above). But h_1 does not appear anywhere else among the constraints, so we can analyze the rest of the program as if this condition were just the literal *true*. This gives the path condition:

$$\text{true} \wedge m''[0] = 101 \tag{4.4}$$

4.3.1.2 Re-Stitching

After decomposing the constraints, our system solves the constraints corresponding to the remainder of the program (excluding the encoding function(s)), as in non-stitched concolic execution, to give a partial input. The re-stitching step builds a complete program input from this partial input by concretely execution encoding functions and their inverses. If the decomposition is correct, such a complete input is guaranteed to exist, but we construct it explicitly so that the exploration process can re-execute the program from the beginning. Once we have found a bug, a complete input confirms (independent of any assumptions about the analysis technique) that the bug is real, allows easy testing on other related samples, and is the first step in creating a working exploit.

For serial decomposition, we are given an input to f_2, and the goal is to find a corresponding input to f_1 that produces that value. This requires access to an inverse function for f_1; we discuss finding one in Section 4.3.2.2. (If f_1 is many-to-one, any inverse will suffice.) For instance, in the example of Figure 4.1, the partial input is a decrypted message, and the full input is the corresponding AES-encrypted message.

For side-condition decomposition, we are given a value for the first part of the input that is processed by f_1. The goal is to find a matching value for the rest of the input that is processed by f_2, such that the predicate p holds. For instance, in Figure 4.1, f_1 corresponds to the function `compute_sha1`, f_2 is the identity function copying the value `m->hash`, and p is the equality predicate. We find such a value by executing f_1 forwards, finding a value related to that value by p, and applying the inverse of f_2. A common special case is that f_2 is the identity function and the predicate p is just equality, in which case we only have to re-run f_1. For Figure 4.1, our tool must simply re-apply `compute_sha1` to each new message.

4.3.1.3 The Functional Perspective

A more abstract perspective on the decomposition our technique performs is to consider the components of the program as if they were pure functions. Of course the real programs we analyze have side-effects: a key aspect of our implementation is to automatically analyze the dependencies between operations to understand which instructions produce values that are read by other instructions. We summarize this structure to understand which operations are independent from others. In this section, we show this independence by modeling a computation as a function that takes as inputs only those values the computation depends on, and whose outputs encompass all of its side effects. This representation is convenient for formally describing the conditions that enable decomposition and re-stitching.

Serial decomposition applies when a program has the functional form $f_2(f_1(i))$ for input i, and the function f_1 (the surjective transformation) is onto: all values that might be used as inputs to f_2 could be produced as outputs of f_1 for some input. Observe that the fact that i does not appear directly as an argument to f_2 implies that f_2 has no direct dependency on the pre-transformed input. For re-stitching, we are given a partial input x_2 in $f(x_2)$, and our tool computes the corresponding full input as $x_1 = f_1^{-1}(x_2)$.

For side-condition decomposition, we say that a predicate p is a free side-condition in a program that has the functional form $f_4(f_3(i_1), p(f_1(i_1), f_2(i_2)))$, where the input is in disjoint parts i_1 and i_2. Here f_2 is a surjective transformation and p is a surjective or right-total relation: for all y there exists an x such that $p(x, y)$ is true. When p is a free side-condition, the effect of decomposition is to ignore f_1, f_2, and p, and analyze inputs i_1 as if the program were $f_4(f_3(i_1), \text{true})$. This gives a partial input x_1 for the computation $f_4(f_3(x_1), \text{true})$. To create a full input, we must also find an additional input x_2 such that $p(f_1(x_1), f_2(x_2))$ holds. Our tool computes this using the formula $x_2 = f_2^{-1}(p^{-1}(f_1(x_1)))$.

4.3.2 Identification

The previous section described the conditions under which decomposition is possible; we next turn to the question of how to automatically identify candidate decomposition sites. Specifically, we first discuss finding encoding functions in Section 4.3.2.1, and then how to find inverses of those functions when needed in Section 4.3.2.2.

4.3.2.1 Identifying Encoding Functions

There are two properties of an encoding function that make it profitable to use for decomposition in our approach. First, the encoding function should be difficult to reason about symbolically. Second, the way the function is used should match one of the decomposition patterns described in Section 4.3.1. Our identification approach is structured to check these two kinds of properties, using a common mechanism of dynamic dependency analysis.

Dynamic dependency analysis. For identifying encoding functions, we perform a trace-based dependency analysis that is a general kind of dynamic tainting. The analysis associates information with each value during execution, propagates that information when values are copied, and updates that information when values are used in an operation to give a new value. Equivalently, this can be viewed as propagating information along edges in the constraint graph (taking advantage of the fact that the execution is a topological-order traversal of that graph). Given the selection of any subset of the program state as a taint source, the analysis computes which other parts of the program state have a data dependency on that source.

Identifying high taint degree. An intuition that partially explains why many encoding functions are hard to reason about is that they mix together constraints related to many parts of the program input, which makes constraint solving difficult. For instance, this is illustrated by a contrast between an encryption function that uses a block cipher in CBC mode, and one that uses a stream cipher. Though the functions perform superficially similar tasks, the block cipher encryption is a barrier to concolic execution because of its high mixing, while a stream cipher is not. Because of the lack of mixing, a constraint solver can efficiently determine that a single plaintext byte can be modified by making a change to the corresponding ciphertext byte. We use this intuition for detecting encoding functions for decomposition: the encoding functions we are interested in tend to mix their inputs.

We can potentially use dynamic dependency analysis to track the dependencies of values on any earlier part of the program state; for instance we have experimented with treating every input to a function as a dependency (taint) source. But for our study, we confine ourselves to using the inputs to the entire program (i.e., from system calls) as dependency sources. To be precise our analysis assigns an identifier to each input byte, and determines, for each value in an execution, which subset of the input bytes it depends on. We call the number of such input bytes the value's *taint degree*. If the taint degree of a byte is larger than a configurable threshold, we refer to it as high-taint-degree. We group together a series of high-taint-degree values in adjacent memory locations as a single buffer; our decomposition applies to a single such buffer.

This basic technique could apply to buffers anywhere in an execution, but we further enhance it to identify functions that produce high-taint-degree buffers as output. This has several benefits: it reduces the number of candidate buffers that need to be checked in later stages, and in cases where the tool needs to later find an inverse of a computation (Section 4.3.2.2), it is convenient to search using a complete function. Our tool heuristically considers a buffer to be an output of a function if it is live at the point in time that a return instruction is executed. Also, we heuristically identify a function that includes the complete encoding functionality by searching for the first high-taint-degree computation that the output buffer depends on, and choosing the function that encloses both this first computation and the output buffer.

In the example of Figure 4.1, the buffers containing the outputs of `aes_cbc_decrypt` and `compute_sha1` would both be found as candidates by this technique, since they both would contain bytes that depend on all of the input bytes (the final decrypted byte, and all of the hash value bytes).

Checking dependence conditions. Values with a high taint degree as identified above are candidates for decomposition because they are potentially problematic for symbolic reasoning. But to apply our technique to them, they must also appear in a proper context in the program to apply our decomposition. Intuitively the structure of the program must be like those in Figure 4.2. To be more precise, we describe (in-)dependence conditions that limit what parts of the program may use values produced by other parts of the program. The next step in our identification approach is to verify that the proper dependence conditions hold (on the observed execution). This checking is needed to avoid improper decompositions, and it also further filters the potential encoding functions identified based on taint degree. If the dependence conditions on the program structure are not satisfied, our technique cannot be applied and thus concolic execution has to be performed in a traditional brute force manner. Although satisfying situations can be very limited, the structures of the program as in Figure 4.2 match common, non-obfuscated usages of encoding functions and thus extend the applications of concolic execution to a larger set of programs.

Intuitively, the dependence conditions require that the encoding function be independent of the rest of the program, except for the specific relationships we expect. For serial decomposition, our tool checks that the input bytes that were used as inputs to the surjective transformation are not used later in the program. For side-condition decomposition, our tool checks that the result of the free side-condition predicate is the only use of the value computed from the main input (e.g., the computed checksum), and that the remaining input (e.g., the expected checksum from a header) is not used other than in the free side-condition. Our tool performs this checking using the same kind of dynamic dependency analysis used to measure taint degree.

In the example of Figure 4.1, our tool checks that the encrypted input to `aes_cbc_decrypt` is not used later in the program (it cannot be, because it is overwritten). It also checks that the hash buffer pointed to by h is not used other than in the `memcmp` on line 11, and that the buffer `m->hash`, containing the expected hash value, is not used elsewhere.

Identifying new encoding functions. The identification step may need to be run in each iteration of the exploration because new encoding functions functions may appear that had not been seen in previous iterations. As an optimization, BitFuzz runs the identification on the first iteration of the exploration, as shown in Figure 4.3, and then, on each new iteration, it checks whether the solver times out when solving any constraint. If it does, it re-runs the identification on the current execution trace.

A graph-based alternative. Our taint-degree dependency analysis can be seen as simple special case of a broader class of algorithms that identify interesting parts of a program from the structure of its data dependency (data-flow) graph. The approach we currently use has efficiency and simplicity advantages because it can operate in one pass over a trace, but in the future we are also interested in exploring more general approaches that explicitly construct the dependency graph. For instance, the interface between the two stages in a serial decomposition must be a cut in the constraint graph, and we would generally expect it to be minimal cut in the sense of the subset partial order. So we can search for candidate serial decompositions by using a maximum-flow-minimum-cut algorithm as in McCamant and Ernst's Flowcheck tool [75].

4.3.2.2 Identifying Inverse Functions

Recall that to re-stitch inputs after serial decomposition, our approach requires the inverses of surjective transformation functions. This requirement is reasonable because surjective functions like decryption and decompression are commonly the inverses of other functions (encryption and compression) that apply to arbitrary data. These functions and their inverses are often used in concert, so their implementations can often be found in the same binaries or in publicly available libraries (e.g., [89, 123]). Thus, we locate relevant inverse functions by searching for possible functions in the code being analyzed as well as in publicly available libraries. If a possible inverse function requires a key or a seed, we supply it with the same key/seed used in the execution trace by the encoding function.

Specifically, we check whether two functions are each others' inverses by random testing. If f and f' are two functions, and for several randomly-chosen x and y, $f'(f(x)) = x$ and $f(f'(y)) = y$, then f and f' are likely inverses of each other over most of their domains. Suppose f is the encoding function we wish to invert. Starting with all the functions from the same binary module that were exercised in the trace, we infer their interfaces using our previous BCR tool [16]. To prioritize the candidates, we use the intuition that the encryption and decryption functions likely have similar interfaces. For each candidate inverse g, we compute a 4-element feature vector counting how many of the parameters are used only for input, only for output, or both, and how many are pointers. We then sort the candidates in increasing order of the Manhattan distances (sum of absolute differences) between their features and those of f.

For each candidate inverse g, we execute $f \circ g$ and $g \circ f$ on k random inputs each, and check whether they both return the original inputs in all cases. If so, we consider g to be the inverse of f. To match the output interface of g with the input interface of f, and vice-versa, we generate missing inputs either according to the semantics inferred by BCR tool[2], or randomly; if there are more outputs than inputs we test each possible mapping. Increasing the parameter k improves the confidence in resulting identification, but the choice of the parameter is not very sensitive: test buffers have enough entropy that even a single false positive is unlikely, but since the tests are just concrete executions, they

[2]BCR tool infers various semantics related to system operations, including field lengths, IP addresses, timestamps, and filenames.

are inexpensive. If we do not find an inverse among the executed functions in the same module, we expand the search to other functions in the binary, in other libraries shipped with the binary, and in standard libraries.

For instance, in the example of Figure 4.1, our tool requires an AES encryption function to invert the AES decryption used by the bot program. In bots it is common for the encryption function to appear in the same binary, since the bot often encrypts its reply messages with the same cipher, but in the case of a standard function like AES we could also find the inverse in a standard library like OpenSSL [89].

Once an inverse function is identified, we use our previous BCR tool to extract the function [16]. The hybrid disassembly technique used by BCR tool extracts the body of the function, including instructions that did not appear in the execution, which is important because when re-stitching a partial input branches leading to those, previously unseen, instructions may be taken. If the inverse function requires a key or a seed, we supply it with the same key/seed used in the execution trace by the encoding function.

4.3.3 Multiple Encoding Layers

If a program has more than one encoding function, we can repeat our approach to decompose the constraints from each encoding function in turn, creating a multi-layered decomposition. The decomposition operates from the outside in, in the order the encoding functions are applied to the input, intuitively like peeling the layers of an onion. For instance, in the example of Figure 4.1, our tool decomposes first the decryption function and then the hash-checking function, finally leaving only the botnet client's command parsing and malicious behavior for exploration.

4.4 Implementation

In this section we provide implementation details for our scaling tool and describe our Internet-in-a-Workstation environment.

4.4.1 Decomposition and Re-stitching of Concolic Execution

We implement our approach as a scaling tool for BitFuzz, a system discussed in Section 2.4. BitFuzz is implemented using the BitBlaze [105] platform for binary analysis, which includes TEMU, an extensible whole-system emulator that implements taint propagation. BitFuzz supports several techniques for vulnerability detection and reports any inputs flagged by these techniques. It detects program termination and invalid memory access exceptions. Executions that exceed a timeout are flagged as potential infinite loops. It also uses TEMU's taint propagation module to identify whether the input (e.g., network data) is used in the program counter or in the size parameter of a memory allocation.

Following the approach introduced in Section 4.3.1.1, our system implements decomposition by making local modifications constraints generated from execution, with some additional optimizations. For serial decomposition, it uses a TEMU extension mechanism called a hook to implement the renaming of symbolic values. As a further optimization, the hook temporarily disables taint propagation inside the encoding function so that no symbolic constraints are generated. To save the work of recomputing a checksum on each iteration in the case of side-condition decomposition, our tool can also directly force the conditional branch implementing the predicate p to take the same direction it did on the original execution.

4.4.2 Internet-in-a-Workstation

We have developed an environment where we can run malware in isolation, without worrying about malicious behavior leaking to the Internet. Many malware programs, e.g., bots, act as network clients that start connections to remote C&C servers. Thus, the input that BitFuzz needs to feed to the program in each iteration is often the response to some request sent by the program.

All network traffic generated by the program, running in the execution monitor, is redirected to the local workstation in a manner that is transparent to the program under analysis. In addition, we have developed two helper tools: a modified DNS server which can respond to any DNS query with a preconfigured or randomly generated, IP address, and a generic replay server. The generic replay server takes as input an XML file that describes a network dialog as an ordered sequence of connections, where each connection can comprise multiple messages in either direction. It also takes as input the payload of the messages in the dialog. Such generic server simplifies the task of setting up different programs and protocols. Given a network trace of the communication we generate the XML file describing the dialog to explore, and give the replay server the seed messages for the exploration. Then, at the beginning of each exploration iteration BitFuzz hands new payload files (i.e., the re-stitched program input) to the replay server so that they are fed to the network client program under analysis when it opens a new connection.

4.5 Experimental Evaluation

This section evaluates our approach by finding bugs in malware that uses complex encoding functions. It demonstrates that our decomposition and re-stitching approach finds some bugs in malware that were not previously found, and that it significantly increases the efficiency of the exploration in other cases. It presents the malware bugs we find and shows that these bugs have persisted in the malware families for long periods of time, sometimes years.

Malware samples. The first column of Table 4.1 presents the four popular families of malware that we have used in our evaluation. Zbot, also known as Zeus and Kollah, is a malware kit that allows malware authors to generate their own variant of password-stealing botnets. Malware from this family is first seen in 2007 and has evolved over time since. The bot program communicates with the C&C server using the HTTP protocol using RC4 encryption on the payload with the key specified by the malware author [46]. MegaD or Ozdok is a prevalent spam botnet that accounted for 35.4% of all spam in the Internet in a December 2008 study and accounts for 15% as of April 2010 [73]. Previous work shows that MegaD's C&C communication is protected using a proprietary block encryption algorithm [18]. Gheg, also known as Tofsee and Mondera, is a spam botnet that can route its spam messages through the victim's ISP server and has 60,000 estimated member as reported in a February 2010 study [63]. It encrypts traffic from the C&C server using proprietary protocol on port 443. Cutwail is a bot program in a Pushdo family that is first seen in 2007 and accounts for 7% of Internet spam from approximately 1.5 million bots in April 2010 [73]. The bot uses RC4 encryption for its C&C communication [107].

All four malware families act as network clients, that is, when run they attempt to connect to a remote C&C server rather than opening a listening socket and await for commands. All four of them encrypt their network communication to avoid signature-based NIDS detection, and make it harder for analysts to reverse-engineer their C&C protocol. In addition to encryption, Zbot also uses an MD5 cryptographic hash function to verify the integrity of a configuration file received from the server. The presence of a hash checksum will frustrate black-box analysis such as random fuzz testing, since most randomly generated input will have an incorrect checksum and so will be rejected at an early stage of the program's input processing.

Experimental setup. For each bot we are given a network trace of the bot communication from which we extract an XML representation of the dialog between the bot and the C&C server, as well as the payload of the network packets in that dialog. This information is

Name	Program size (KB)	Input size (bytes)	# Instruction ($\times 10^3$)	Decryption or checksum/hash		Runtime (sec)
				Algorithm	Maximum taint degree	
Zbot	126.5	5269	1307.3	RC4-256	1	92
				MD5	4976	
MegaD	71.0	68	4687.6	64-bit block cipher	8	105
Gheg	32.0	271	84.5	8-bit stream cipher	128	5
Cutwail	50.0	269	23.1	byte-based cipher	1	2

Table 4.1: Summary of the Applications on Which We Performed Identification of Encoding Functions.

needed by the replay server to provide the correct sequence of network packets to the bot during exploration. For example, this is needed for MegaD where the response sent by the replay server comprises two packets that need to be sent sequentially but cannot be concatenated together due to the way that the bot reads from the socket. As a seed for the exploration we use the same content observed in the dialog captured in the network trace. Other seeds can alternatively be used. Although our setup can support exploring multiple connections, currently, we focus the exploration on the first connection started by the bot.

For the experiments we ran BitFuzz on a 3GHz Intel Core 2 Duo Linux workstation with 4GB of RAM running Ubuntu Server 9.04. The emulated guest system where the malware program runs is a Microsoft Windows XP SP3 image with 512MB of emulated RAM.

In each of our experiments, our system only used a malware sample and a network trace as its inputs. In the experiments that require inverse functions, our system was able to successfully extract the inverse functions from execution traces. However, successful extraction is actually not guaranteed. This limitation will be discussed later in Section 4.6.2.

4.5.1 Identification of Encoding Functions and Their Inverses

The first step in our approach is to identify the encoding functions. The identification of the encoding functions happens on the execution trace produced by the seed at the beginning of the exploration. We set the taint degree threshold to 4, so that any byte that has been generated from 5 or more input bytes is flagged. Table 4.1 summarizes the results. The identification finds an encoding function in three of the four samples: Gheg, MegaD, and Zbot. For Cutwail, no encoding function is identified. The reason for this is that Cutwail's cipher is simple and does not contain any mixing of the input, which is the property that our encoding function identification technique detects. Without input mixing the constraints generated by the cipher are not complex to solve. We show this in the next section. In addition, Cutwail's trace does not contain any checksum functions.

For Zbot, the encoding function flagged in the identification corresponds to the MD5 checksum that it uses to verify the integrity of the configuration file it downloads from the C&C server. In addition to the checksum, Zbot uses the RC4 cipher to protect its communication, which is not flagged by our identification technique described in Section 4.3.2.1. This happens because RC4 is a stream cipher that does no mixing of the input, i.e., it does not use input or output bytes to update its internal state. The input is simply combined with a pseudo-random keystream using bit-wise exclusive-or. Since the keystream is not derived from the input but from a key in the data section, it is concrete for the solver. Thus, the solver only needs to invert the exclusive-or computation to generate an input, which means that RC4 introduces no hard-to-solve constraints. As a result, we do not perform decomposition and restitching on the program conditions that come from the RC4 cipher.

For the other two samples (Gheg and MegaD) the encoding function flagged by the identification corresponds to the cipher. MegaD uses a 64-bit block cipher, which mixes 8 bytes from the input before combining them with the key. Gheg's cipher uses a one-byte key that is combined with the first input byte to produce a one-byte output that is used also as key to encode the next byte. This process repeats and the mixing (taint degree) of each new output byte increases by one. Neither Gheg nor MegaD uses a checksum.

Once the encoding functions have been identified, BitFuzz introduces new symbols for the outputs of those encoding functions, effectively decomposing the constraints in the execution into two sets and ignoring the set of hard-to-solve constraints introduced by the encoding function.

The results of our encoding function identification, for the first iteration of the exploration, are summarized in Table 4.1, which presents on the left the program name and program size, the size of the input seed, and the number of instructions in the execution trace produced by the seed. The decryption or checksum column describes the algorithm type and the maximum taint degree the algorithm produces in the execution. We display all decryption and checksum algorithms known to appear in the execution, regardless of whether our tool considers them encoding functions (i.e., having Maximum taint degree of 5 or higher) or not. The rightmost column shows the runtime of the identification algorithm, which varies from a few seconds to close to two minutes. Because the identification is reused over a large number of iterations, the amortized overhead is even smaller.

Identifying the inverse functions. For Gheg and MegaD, BitFuzz needs to identify the inverse of the decryption function so that it can be used to re-stitch the inputs into a new program input for another iteration. (The encryption function for MegaD is the same one identified in previous work [16]; we use it to check the accuracy of our new identification approach.)

As described in Section 4.3.2.2, BitFuzz extracts the interface of each function in the execution trace that belongs to the same module as the decoding function, and then prioritizes them by the similarity of their interface to the decoding function. For both Gheg and MegaD, the function with the closest prototype is the encryption function, as our tool confirms by random testing with $k = 10$ tests. These samples illustrate the common pattern of a matching encryption function being included for two-way communication, so we did not need to search further afield for an inverse.

Name	Vulnerability type	Disclosure public identifier	Encoding functions	Search time (min.)	
				scaled	baseline
Zbot	Null dereference	OSVDB-66499 [94]	checksum	17.8	>600
	Infinite loop	OSVDB-66500 [93]	checksum	129.2	>600
	Buffer overrun	OSVDB-66501 [92]	checksum	18.1	>600
MegaD	Process exit	n/a	decryption	8.5	>600
Gheg	Null dereference	OSVDB-66498 [91]	decryption	16.6	144.5
Cutwail	Buffer overrun	OSVDB-66497 [90]	none	39.4	39.4

Table 4.2: Description of the Bugs Our System Finds in Malware. The column "scaled" shows the results from the BitFuzz system including our decomposition and re-stitching techniques, while the "baseline" column gives the results with these techniques disabled. ">600" means we run the tool for 10 hours and it is yet to find the bug.

4.5.2 Decomposition vs. Non-Decomposition

In this section we compare the number of bugs found by BitFuzz when it uses decomposition and re-stitching, which we call *full* BitFuzz, and when it does not, which we call *vanilla* BitFuzz. Full BitFuzz uses the identified decoding functions to decompose the constraints into two sets, one with the constraints introduced by the decryption/checksum function and the other with the remaining constraints after that stage. In addition, each iteration of MegaD and Gheg uses the inverse function to re-stitch the inputs into a program input. Vanilla BitFuzz is comparable to basic previous execution tools. In both full and vanilla cases, BitFuzz detects bugs using the techniques described in Section 4.4.

In each iteration of its exploration, BitFuzz collects the execution trace of the malware program starting from the first time it receives network data. It stops the trace collection when the malware program sends back a reply, closes the communication socket, or a bug is detected. If none of those conditions is satisfied the trace collection is stopped after 2 minutes. For each collected trace, BitFuzz analyzes up to the first 200 input-dependent control flow branches and automatically generates new constraints that would explore new paths in the program. It then queries STP to solve each generated set of constraints, uses the solver's response to generate a new input, and adds it to the pool of inputs to test on future iterations. Because constraint solving can take a very long time without yielding a meaningful result, BitFuzz discards a set of constraints if STP runs out of memory or exceeds a 5-minute timeout for constraint solving.

We run both vanilla and full BitFuzz for 10 hours and report the bugs found, which are summarized in Table 4.2. Detailed descriptions of the bugs follow in Section 4.5.3. We break the results in Table 4.2 into three categories. The first category includes Zbot and MegaD for which full BitFuzz finds bugs but Vanilla BitFuzz does not. Full BitFuzz finds a total of four bugs, three in Zbot and one in MegaD. Three of the bugs are found in under 20 minutes and the second Zbot bug is found after 2 hours. Vanilla BitFuzz does not find any bugs in the 10-hour period. This happens due to the complexity of the constraints being introduced by the encoding functions. In particular, using full BitFuzz the 5-minute timeout for constraint solving is never reached and STP never runs out of memory, while using vanilla BitFuzz more than 90% of the generated constraints result in STP running out of memory.

The second category comprises Gheg for which both vanilla and full BitFuzz find the same bug. Although both tools find the same bug, we observe that vanilla BitFuzz requires almost ten times as long as full BitFuzz to do so. The cipher used by Gheg uses a one-byte hardcoded key that is combined with the first input byte using bitwise exclusive-or to produce the first output byte, that output byte is then used as key to encode the second byte also using bitwise exclusive-or and so on. Thus, the taint degree of the first output byte is one, for the second output byte is two and so on until the maximum taint degree of 128 shown in Table 4.1. The high maximum taint degree makes it harder for the solver to solve and explains why vanilla BitFuzz takes much longer than full BitFuzz to find the bug. Still, the constraints induced by the Gheg cipher are not as complex as the ones induced by the Zbot and MegaD ciphers and the solver eventually finds solutions for them. This case shows that even in cases where the solver will eventually find a solution, using decomposition and re-stitching can significantly improve the performance of the exploration.

The third category comprises Cutwail for which no encoding functions with high taint degree are identified and thus vanilla BitFuzz and full BitFuzz are equivalent.

In summary, full BitFuzz using decomposition and re-stitching clearly outperforms vanilla BitFuzz. Full BitFuzz finds bugs in cases where vanilla BitFuzz fails to do so due to the complexity of the constraints induced by the encoding functions. It also improves the performance of the exploration in other cases were the encoding constraints are not as complex and will eventually be solved.

4.5.3 Malware Vulnerabilities

In this section we present the results of our manual analysis to understand the bugs discovered by BitFuzz and our experiences reporting the bugs.

Zbot. BitFuzz finds three bugs in Zbot. The first one is a null pointer dereference. One of the C&C messages contains an array size field, which the program uses as the size parameter in a call to `RtlAllocateHeap`. When the array size field is larger than the available memory left in its local heap, the allocation returns a null pointer. The return value of the allocation is not checked by the program, which later attempts to write to the buffer, crashing when it tries to dereference the null pointer.

The second bug is an infinite loop condition. A C&C message comprises of a sequence of blocks. Each block has a 16-byte header and a payload. One of the fields in the header represents the size of the payload, s. When the trojan program finishes processing a block, it iteratively moves to the next one by adding the block size, $s + 16$, to a cursor pointer. When the value of the payload size is $s = -16$, the computed block size becomes zero, and the trojan keeps processing the same block over and over again.

The last bug is a stack buffer overrun. As mentioned above, a C&C message comprises of a sequence of blocks. One of the flags in the block header determines whether the block payload is compressed or not. If the payload is compressed, the trojan program decompresses it by storing the decompressed output into a fixed-size buffer located on the stack. When the length of the decompressed payload is larger than the buffer size, the program will write beyond the buffer. If the payload is large enough, it will overwrite a function return address and can eventually lead to control flow hijacking. This vulnerability is exploitable and we have successfully crafted a C&C message that exploits the vulnerability and hijacks the execution of the malware.

MegaD. BitFuzz finds one input that causes the MegaD bot to exit cleanly. We analyzed this behavior using the MegaD grammar produced by previous work [18] and found that the bug is present in the handling of the *ping* message (type `0x27`). If the bot receives a ping message and the bot identifier (usually set by a previously received C&C message) has not been set, then it sends a reply *pong* message (type `0x28`) and terminates. This behavior highlights the fact that, in addition to bugs, our stitched concolic execution can also discover C&C messages that cause the malware to cleanly exit (e.g., kill commands), if those commands are available in the C&C protocol. These messages cannot be considered bugs but can still be used to disable the malware. They are specially interesting because they may have been designed to completely remove all traces of the malware running in the compromised host. In addition, their use could raise fewer ethical and legal questions than the use of an exploit would.

Gheg. BitFuzz finds one null pointer dereference bug in Gheg. The bug is similar to the one in Zbot. One of the C&C messages contains an array size field, whose value is multiplied by a constant (0x1e8) and the result used as the size parameter in a call to `RtlAllocateHeap`. The return value of the allocation is not checked by the program and the program later writes into the allocated buffer. When the array size field value is larger than the available memory in its local heap, the allocation fails and a null pointer is returned. The program fails to check that the returned value is a null pointer and tries to dereference it.

Cutwail. BitFuzz finds a buffer overrun bug that leads to an out-of-bounds write in Cutwail. One of the received C&C messages contains an array. Each record in the array has a length field specifying the length of the record. This field is used as the size parameter in a call to `RtlAllocateHeap`. The returned pointer is appended to a global array that can only hold 50 records. If the array in the received message has more than 50 records, the 51^{st} record will be written outside the bounds of the global array. Near the global array, there exists a pointer to a private heap handle and the out-of-bounds write will overwrite this pointer. Further calls to `RtlAllocateHeap` will then attempt to access the malformed heap handle, and will lead to heap corruption and a crash.

Reporting the bugs. We reported the Gheg bug to the editors of the Common Vulnerabilities and Exposures (CVE) database [38]. Our suggestion was that vulnerabilities in malware should be treated similarly to vulnerabilities in commercial or open source programs, of course without reporting back to the developers. However, the CVE editors felt that malware vulnerabilities were outside the scope of their database. Subsequently, we reported the Gheg vulnerability to the Open Source Vulnerability Database (OSVDB) moderators who accepted it. Since then, we have reported all other vulnerabilities except the MegaD one, which may be considered intended functionality by the botmaster. Table 4.2 presents the public identifiers for the disclosed vulnerabilities. We further address the issue of disclosing malware vulnerabilities in Section 4.6.

4.5.4 Bug Persistence over Time

Bot binaries are updated very often to avoid detection by anti-virus tools. One interesting question is how persistent over time are the bugs found by BitFuzz. To evaluate this, we retest our crashing inputs on other binaries from the same malware families. Table 4.3 shows all the variants, with the shaded variants corresponding to the ones explored by BitFuzz and mentioned in Table 4.1.

We replay the input that reproduces the bug BitFuzz found on the shaded variant on the rest of variants from the same family. As shown, the bugs are reproducible across all the

Family	MD5	First seen	Reported by
Zbot	0bf2df85*7f65	Jun-23-09	Prevx
	1c9d16db*7fc8	Aug-17-09	Prevx
	7a4b9ceb*77d6	Dec-14-09	ThreatExpert
MegaD	700f9d28*0790	Feb-22-08	Prevx
	22a9c61c*e41e	Dec-13-08	Prevx
	d6d00d00*35db	Feb-03-10	VirusTotal
	09ef89ff*4959	Feb-24-10	VirusTotal
Gheg	287b835b*b5b8	Feb-06-08	Prevx
	edde4488*401e	Jul-17-08	Prevx
	83977366*b0b6	Aug-08-08	ThreatExpert
	cdbd8606*6604	Aug-22-08	Prevx
	f222e775*68c2	Nov-28-08	Prevx
Cutwail	1fb0dad6*1279	Aug-03-09	Prevx
	3b9c3d65*07de	Nov-05-09	Prevx

Table 4.3: Bug Reproducibility Across Different Malware Variants. The shaded variants are the ones used for exploration.

variants we tested. This means for instance that the MegaD bug has been present for at least two years (the time frame covered by our variants). In addition, the MegaD encryption and decryption functions (and the key they use), as well as the C&C protocol have not changed, or barely evolved, through time. Otherwise the bug would not be reproducible in older variants. The results for Gheg are similar. The bug reproduces across all Gheg variants, although in this case our most recent sample is from November 2008. Note that, even though the sample is relatively old it still works, meaning that it still connects to a C&C server on the Internet and sends spam. For Zbot, all three bugs reproduce across all variants; this means they have been present for at least 6 months. These results are important because they demonstrate that there are components in bot software, such as the encryption functions and C&C protocol grammar, that tend to evolve slowly over time and thus could be used to identify the family to which an unknown binary belongs, one widespread problem in malware analysis.

4.6 Discussion

In light of our results, this section provides additional discussion on the applications for the discovered bugs and associated ethical considerations. Then, it presents a potential scenario for using the discovered bugs, and describes some limitations of our approach.

4.6.1 Applications and Ethical Considerations

Malware vulnerabilities could potentially be used in different “benign” applications such as remediating botnet infestations, for malware genealogy since we have shown that the bugs persist over long periods of time, as a capability for law enforcement agencies, or as a strategic resource in state-to-state cyberwarfare [95]. However, their use raises important ethical and legal questions. For example, there may be a danger of significant negative consequences, such as adverse effects to the infected machines. Also, it is unclear which legal entity would perform such remediation, and whether currently there exists any entity with the legal right to take such action. On the other hand, having a potential avenue for cleanup and not making use of it also raises some ethical concerns since if such remediation were effective, it would be a significant service to the malware’s future third-party victims (targets of DDoS attacks, spam recipients, etc.). Such questions belong to recent and ongoing discussions about ethics in security research (e.g., [43]) that have not reached a firm conclusion.

Malware vulnerabilities could also be used for malign purposes. For instance, there are already indications that attackers are taking advantage of known vulnerabilities in web interfaces used to administer botnets to hijack each other’s botnets [40]. This raises concerns about disclosing such bugs in malware. In the realm of vulnerabilities in benign software, there has been significant debate on what disclosure practices are socially optimal and there is a partial consensus in favor of some kind of “responsible disclosure” that gives authors a limited form of advance notice. However, it is not clear what the analogous best practice for malware vulnerabilities should be. We have faced this disclosure issue when deciding whether to publicly disclose the vulnerabilities we found and to which extent we should describe the vulnerabilities. We hope the vulnerabilities reported here do provide an appropriate level of details.

Potential application scenario. While we have not used our crashing inputs on bots in the wild, here we hypothetically discuss one possible scenario of how one might do so. The malware programs we analyze start TCP connections with a remote C&C server. To exploit the vulnerabilities we have presented, we need to impersonate the C&C server and feed inputs in the response to the initial request from the malware program. This scenario often happens during a botnet takedown, in which law enforcement or other responding entities identify the IP addresses and DNS names associated with the C&C servers used by a botnet, and appeal to relevant ISPs and registrars to have them de-registered or redirected to the responders. The responders can then impersonate the C&C server: one common choice is a *sinkhole server* that collects statistics on requests but does not reply. But such responders are also in a position to perform more active communication with bots, and for instance vulnerabilities like the ones we present could be used for cleanup if the botnet does not support cleanup via its normal protocol. For example, such a scenario happened recently during the attempted MegaD takedown by FireEye [77]. For a few days FireEye ran a sinkhole server that received the C&C connections from the bots. This sinkhole server was later handed to the Shadowserver Foundation [103].

4.6.2 Limitations

We have found our techniques to be quite effective against the current generation of malware. But since malware authors have freedom in how they design encoding functions, and an incentive to avoid analysis of their programs, it is valuable to consider what measures they might take against analysis.

Preventing access to inverses. To stitch complete inputs in the presence of a surjective transformation, our approach requires access to an appropriate inverse function: for instance, the encryption function corresponding to a decryption function. So far, we have been successful in finding such inverses within the malware binary. If the inverses were not present in the binary, we could find them from standard sources. However, these approaches could be thwarted if malware authors made different choices of cryptographic algorithms. For instance, malware authors could design their protocols using asymmetric (public-key) encryption and digital signatures. Since we would not have access to the private key used by the C&C server, we could not forge the signature in the messages sent to the bot. We could still use our decomposition and re-stitching approach to find bugs in malware, because the signature verification is basically a free side-condition that can be ignored. However, we could only build an exploit for our modified bot, as other bots will verify the (incorrect) signature in the message and reject it. Currently, most malware do not use public-key cryptography, but that may change. In the realm of symmetric encryption, malware authors could deploy different non-standard algorithms for the server-to-bot and bot-to-server directions of communication: though not theoretically infeasible, the construction of an encryption implementation from a binary decryption implementation might be challenging to automate. For instance, Kolbitsch et al. [67] faced such a situation in recreating binary updates for the Pushdo trojan, which was feasible only because the decryption algorithm used was weak enough to be inverted by brute force for small plaintexts.

Obfuscating encoding functions. Malware authors could potentially keep our system from finding encoding functions in binaries by obfuscating them. General purpose packing is not an obstacle to our dynamic approach, but more targeted kinds of obfuscation would be a problem. For instance, our current implementation recognizes only standard function calls and returns, so if a malware author rewrote them using non-standard instructions our tool would require a corresponding generalization to compensate. Further along the arms race, there are also fundamental limitations arising from our use of a dynamic dependency analysis, similar to the limitations of dynamic taint analysis [24].

4.7 Related Work

One closely related recent project is Wang et al.'s TaintScope system [114]. Our goals partially overlap with theirs in the area of checksums, but our work differs in three key aspects. First, Wang et al.'s techniques do not apply to decompression or decryption. Second, TaintScope performs exploration based on taint-directed fuzzing [49], while our system harnesses the full generality of concolic execution. Third, Wang et al. evaluate their tool only on benign software, while we perform the first automated study of vulnerabilities in malware.

The encoding functions we identify within a program can also be extracted from a program to be used elsewhere. The Binary Code Reuse [16] and Inspector Gadget [67] systems can be used to extract encryption and checksum functionalities, including some of the same ones our tool identifies, for applications such as network defense. Our application differs in that our system can simply execute the code in its original context instead of extracting it. Inspector Gadget [67] can also perform so-called gadget inversion, which is useful for the same reasons as we search for existing inverse functions. However, their approach does not work on strong cryptographic functions.

Previous work in protocol reverse engineering has used alternative heuristics to identify cryptographic operations in malware binaries. For instance ReFormat [115] and Dispatcher [18] propose detecting such functions by measuring the ratio of arithmetic and bitwise instructions to other instructions. Our use of taint degree as a heuristic is more specifically motivated by the limitations of concolic execution: for instance a simple stream cipher would be a target of the previous approaches but is not for our approach.

Decomposition is a broad class of techniques in program analysis and verification, but most previous decomposition techniques are symmetric in the sense that each of the sub-components of the program are analyzed in a similar way, while a key aspect of our approach is that different components are analyzed differently. In analysis and verification, decomposition at the level of functions, as in systems like Saturn [118], is often called a compositional approach. In the context of tools based on concolic execution, Godefroid [51] proposes a compositional approach that performs concolic execution separately on each function in a program. Because this is a symmetric technique, it would not address our problem of encoding functions that are too complex to analyze even in isolation. More similar to our approach is grammar-based fuzzing [17, 52], which is an instance of serial decomposition. However parsers require different specialized techniques than encoding functions.

4.8 Conclusion

We have presented a new approach, stitched concolic execution, to allow analysis in the presence of functionality that would otherwise be difficult to analyze. Our techniques for automated identification, decomposition, and re-stitching allow our system to bypass functions like decryption and checksum verification to find bugs in core program logic. Specifically, these techniques enable the first automated study of vulnerabilities in malware. Our BitFuzz tool finds 6 unique bugs in 4 prevalent malware families. These bugs can be triggered over the network to terminate or take control of a malware instance. These bugs have persisted across malware revisions for months, and even years. There are still many unanswered questions about the applications and ethical concerns surrounding malware vulnerabilities, but our results demonstrate that vulnerabilities in malware are an important security resource that should be the focus of more research in the future.

Chapter 5

Model-assisted Concolic Execution

5.1 Introduction

In this chapter, we propose a new technique for exploring the program's state-space. The technique explores the program execution space automatically by combining exploration with learning of an abstract model of program's state space. More precisely, it alternates (1) concolic execution to explore the program's state-space, and (2) the L^* [2] online learning algorithm to construct high-level models of the state-space. Such abstract models, in turn, guide further search. In contrast, the prior state-space exploration techniques treat the program as a flat search-space, without distinguishing states that correspond to important input processing events.

Upon closer examination of concolic execution, we identified two of its weaknesses that can be improved. First, concolic execution has no high-level information about the structure of the overall program state-space. Thus, it has no way of knowing how close (or how far) it is from reaching important states in the program and is likely to get stuck in local state-subspaces, such as loops. Second, unlike decision procedures that learn search-space pruning lemmas from each iteration (e.g., [121]), concolic execution only tracks the most promising path prefix for the next iteration [53], but does not learn in the sense that information gathered in one iteration is used either to prune the search-space or to get to interesting states faster in later iterations.

These two insights led us to develop an approach — Model-assisted Concolic Execution (MACE) — that learns from each iteration and constructs a finite-state model of the search-space. We primarily target applications that maintain an ongoing interaction with its environment, like servers and web services, for which a finite-state model is frequently a suitable abstraction of the communication protocol, as implemented by the application. At the same time, we both learn the protocol model and exploit the model to guide the search.

MACE relies upon concolic execution to discover the protocol messages, uses a special filtering component to select messages over which the model is learned, and guides further search with the learned model, refining it as it discovers new messages. Those three components alternate until the process converges, automatically inferring the protocol state machine and exploring the program's state-space.

We have implemented our approach and applied it to four server applications (two SMB and two RFB implementations). MACE significantly improved the line coverage of the analyzed applications, and more importantly, discovered four new vulnerabilities and three known ones. One of the discovered vulnerabilities received Gnome's "Blocker" severity, the highest severity in their ranking system meaning that the next release cannot be shipped without a fix.

5.2 Related Work

Model-guided testing has a long history. The hardware testing community has developed modeling languages, like SystemVerilog, that allow verification teams to specify input constraints that are solved with a decision procedure to generate random inputs. Such inputs are randomized, but adhere to the specified constraints and therefore tend to reach much deeper into the tested system than purely random tests. Constraint-guided random test generation is a staple of hardware testing. The software community developed its own languages, like Spec# [7], for describing abstract software models. Such models can be used effectively as constraints for generating tests [110], but have to be written manually, which is both time consuming and requires a high level of expertise.

Grammar inference (e.g., [42]) promises automatic inference of models, and has been an active area of research in security, especially applied to protocol inference. Comparetti et al. [29] infer incomplete (possibly missing transitions) protocol state machines from messages collected by observing network traffic. To reduce the number of messages, they cluster messages according to how similar the messages are and how similar their effects are on the execution. Comparetti et al. show how the inferred protocol models can be used for fuzzing. Our work shares similar goals, but features a few important differences. First, MACE iteratively refines the model using concolic execution for the state-space exploration. Second, rather than filtering out individual messages through clustering of individual messages, we look at the entire sequences. If there is a path in the current state machine that produces the same output sequence, we discard the corresponding input sequence. Otherwise, we add all the input messages to the set used for inferring the state machine in the next iteration. Third, rather than using the inferred model for fuzzing, we use the inferred model to initialize state-space exploration to a desired state, and then run concolic execution from the initialized state.

In the work [27] prior to ours, the authors proposed an alternative protocol state machine inference approach. There they assume the end users would provide abstraction functions that abstract concrete input and output messages into an abstract alphabet, over which they infer the protocol. Designing such abstraction functions is sometimes nontrivial and requires multiple iterations, especially for proprietary protocols, for which specifications are not available. In our approach, we drop the requirement for user-provided input message abstraction, but we do require a user-provided output message abstraction function. The output abstraction function determines the granularity of the inferred abstraction. The right granularity of abstraction is important for guiding state-space exploration, because too fine-grained abstractions tend to be too expensive to infer automatically, and too abstract ones fail to differentiate interesting protocol states. Furthermore, the prior work is a purely black-box approach, while in our approach we do code analysis at the binary level in combination with grammatical inference.

In this thesis, we analyze implementations of protocols for which the source code or specifications are available. However, MACE could also be used for inference of proprietary protocols and for state-exploration of closed-source third-party binaries. In that case, the users would need to rely upon the prior research to construct a suitable output abstraction function. The first step in constructing a suitable output abstraction function is understanding the message format. Cui et al. [35, 36] and Caballero et al. [20] proposed approaches that could be used for that purpose. Further, any automatic protocol inference technique has to deal with encryption. In our study, we simply configure the analyzed server applications so as to disable encryption, but that might not be an option when inferring a proprietary protocol. Our technique of decomposition and restitching discussed in Chapter 4, though not yet, could be integrated with MACE to deal with encryption. The work of Caballero et al. [18] and Wang et al. [115] also addresses potential techniques to automatic reverse-engineering of encrypted messages.

Software model checking tools, like SLAM [6] and Blast [58], incrementally build predicate abstractions of the analyzed software, but such abstractions are very different from the models inferred by the protocol inference techniques [28, 29]. Such abstractions closely reflect the control-flow structure of the software from which they were inferred, while our inferred models are more abstract and tend to have little correlation with the low-level program structure. Further, depending on the inference approach used, the inferred models can be minimal (like in our work), which makes guidance of state-space exploration techniques more effective.

The Synergy algorithm [55] combines model-checking and concolic execution to try to cover all abstract states of a program. Our work has no ambition to produce proofs, and we expect that our approach could be used to improve the concolic execution part of Synergy and other algorithms that use concolic execution as a component.

The Ketchum approach [59] combines random simulation to drive a hardware circuit into an interesting state (according to some heuristic), and performs local bounded model checking around that state. After reaching a predefined bound, Ketchum continues random simulation until it stumbles upon another interesting state, where it repeats bounded model checking. Ketchum became the key technology behind Magellan$^{\text{TM}}$, one of the most successful semi-formal hardware test generation tools. MACE has similar dynamics, but the components are very different. We use the L^* [2] finite-state machine inference algorithm to infer a high-level abstract model and declare all the states in the model as interesting, while Ketchum picks interesting states heuristically. While Ketchum uses random simulation, we drive the analyzed software to the interesting state by finding the shortest path in the abstract model. Ketchum explores the vicinity of interesting states via bounded model checking, while we start concolic execution from the interesting state.

5.3 Problem Definition and Overview

We begin this section with the problem statement and a list of assumptions that we make. Next, we discuss possible applications of MACE. At the end of this section, we introduce the concepts and notation that will be used throughout the chapter.

5.3.1 Problem Statement

We have three mutually supporting goals. First, we wish to automatically infer an abstract finite-state model of a program's interaction with its environment, i.e., a protocol as implemented by the program. Second, once we infer the model, we wish to use it to guide a combination of concrete and symbolic execution in order to improve the state-space exploration. Third, if the exploration phase discovers new types of messages, we wish to refine the abstract model, and repeat the process.

There are two ways to refine the abstract finite-state model; by adding more states, and by adding more messages to the state machine's input (or output) alphabet, which can

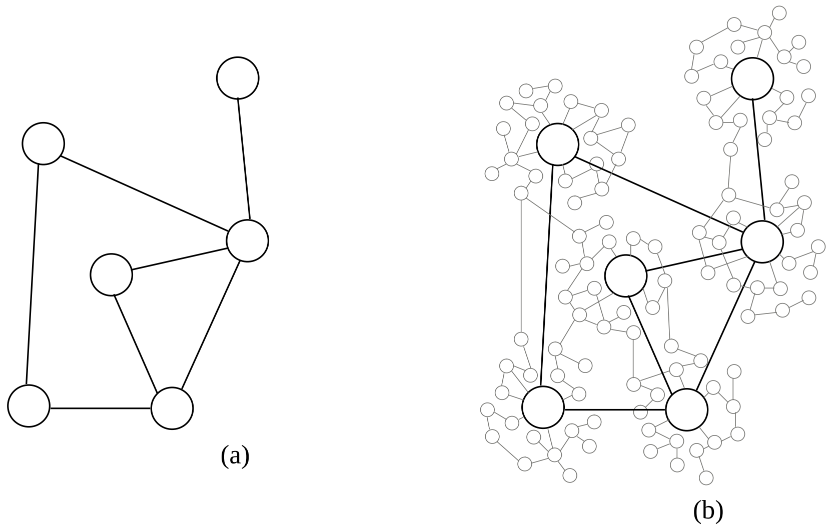


Figure 5.1: An Abstract Rendition of the MACE State-Space Exploration. The figure on the left shows an abstract model, i.e., a finite-state machine, inferred by MACE. The figure on the right depicts clusters of concrete states of the analyzed application, such that clusters are abstracted with a single abstract state. We infer the abstract model with L^*, initialize the analyzed application to the desired state, and then use the state-space exploration component of MACE to explore the concrete clusters of states.

result in inference of new transitions and states. Black box inference algorithms, like L^* [2], infer a state machine over a fixed-size alphabet by iteratively discovering new states. Such algorithms can be used for the first type of refinement. Any traditional program state-space exploration technique could be used to discover new input (or output) messages, but adding all the messages to the state machine's alphabets would render the inference computationally infeasible. Thus, we also wish to find an effective way to reduce the size of the alphabet, without missing states during the inference.

The constructed abstract model can guide the search in many ways. The approach we take is to use the abstract model to generate a sequence of inputs that will drive the abstract model and the program to the desired state. After the program reaches the desired state, we explore the surrounding state-space using a combination of symbolic and concrete execution. Through such exploration, we might visit numerous states that are all abstracted with a single state in the abstract model and discover new inputs that can refine the abstract model. Figure 5.1 illustrates the concept.

In our work, we make a few assumptions:

Determinism We assume the analyzed program's communication with its environment is deterministic, i.e., the same sequence of inputs always leads to the same sequence of outputs and the same state. In practice, programs can exhibit some non-determinism, which we are abstracting away. For example, the same input message could produce two different outputs from the same state. In such a case, we put both output messages in the same equivalence class by adjusting our output abstraction (see below).

Resettability We assume the analyzed program can be easily reset to its initial state. The reset may be achieved by restarting the program, re-initializing its environment or variables, or simply initiating a new client connection. In practice, resetting a program is usually straightforward, since we have a complete control of the program.

Output Abstraction Function We assume the existence of an output abstraction function that abstracts concrete response (output) messages from the server into an abstract set of messages (alphabet) used for state machine inference. In practice, this assumption often reduces to manually identifying which sub-fields of output messages will be used to distinguish output message types. The output alphabet, in MACE, determines the granularity of abstraction.

5.3.2 Applications

The primary intended application of MACE is state-space exploration of programs communicating with their environment through a protocol, e.g., networked applications. We use the inferred protocol state machine as a map that tells us how to quickly get to a particular part of the search-space. In comparison, model checking and concolic execution approaches consider the application's state-space flat, and do not attempt to exploit the structure in the state machine of the communication protocol through which the application communicates with the world. Other applications of MACE include proprietary protocol inference, extension of the existing protocol test suites, conformance checking of different protocol implementations, and fingerprinting of implementation differences.

5.3.3 Preliminaries

Following our prior work [27], we use *Mealy machines* [76] as abstract protocol models. Mealy machines are natural models of protocols because they specify transition and output functions in terms of inputs. Mealy machines are defined as follows:

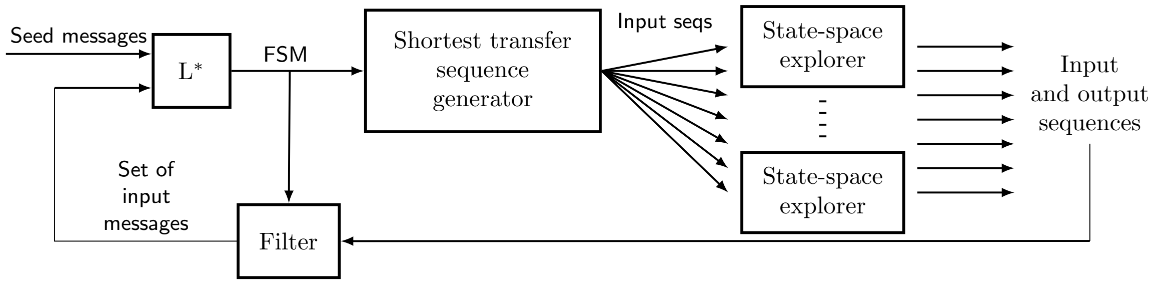


Figure 5.2: The MACE Approach Diagram. The L^* algorithm takes in the input and output alphabets, over which it infers a state-machine. L^* sends queries and receives responses from the analyzed application, which is not shown in the figure. The result of inference is a finite-state machine (FSM). For every state in the inferred state machine, we generate a shortest transfer sequence (Section 5.3.3) that reaches the desired state, starting from the initial state. Such sequences are used to initialize the state-space explorer, which runs concolic execution after the initialization. The state-space explorers run the analyzed application (not shown) in parallel.

Definition 1 (Mealy Machine)**.** *A Mealy machine, M, is a six-tuple $(Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$, where Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, Σ_I is a finite set of input symbols (i.e., the input alphabet), Σ_O is a finite set of output symbols (i.e., the output alphabet), $\delta : Q \times \Sigma_I \longrightarrow Q$ is the transition relation, and $\lambda : Q \times \Sigma_I \longrightarrow \Sigma_O$ is the output relation.*

We extend the δ and λ relations to sequences of messages $s = s_0 \cdot \cdot \cdot_1 \cdots s_{n-1} \in \Sigma_I^*$ as usual, e.g., $\delta(q, s_0 \cdot s_1 \cdot s_2) = \delta(\delta(\delta(q, s_0), s_1), s_2)$ and $\lambda(q, s_0 \cdot s_1 \cdot s_2) = \lambda(q, s_0) \cdot \lambda(\delta(q, s_0), s_1) \cdot \lambda(\delta(q, s_0 \cdot s_1), s_2)$. To denote sequences of input (resp. output) messages we will use lower-case letters s, t (resp. o). For $s \in \Sigma_I^*, m \in \Sigma_I$, the *length* $|s|$ is defined inductively: $|\epsilon| = 0, |s \cdot m| = |s| + 1$, where ϵ is the empty sequence. If for some state machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ and some state $q \in Q$ there is $s \in \Sigma_I^*$ such that $\delta(q_0, s) = q$, we say there is a path from q_0 to q, i.e., that q is reachable from the initial state, denoted $q_0 \xrightarrow{*} q$. Since L^* infers minimal state machines, all states in the abstract model are reachable [2]. In general, each state could be reachable by multiple paths. For each state q, we (arbitrary) pick one of the shortest paths formed by a sequence of input messages s, such that $q_0 \xrightarrow{s} q$, and call it a *shortest transfer sequence*.

Our search process discovers numerous input and output messages, and using all of them for the model inference would not scale. Thus, we heuristically discard redundant input messages, defined as follows:

Definition 2 (Redundant Input Symbols)**.** *Let $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ be a Mealy machine. A symbol $m \in \Sigma_I$ is said to be redundant if there exists another symbol, $m' \in \Sigma_I$, such that $m \neq m'$ and $\forall q \in Q \ . \ \lambda(q, m) = \lambda(q, m') \wedge \delta(q, m) = \delta(q, m')$.*

We say that a Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ is complete iff $\delta(q, i)$ and $\lambda(q, i)$ are defined for every $q \in Q$ and $i \in \Sigma_I$. MACE infers complete Mealy machines. There is also another type of completeness — the completeness of the input and output alphabet. MACE cannot guarantee that the input alphabet is complete, meaning that it might not discover some types of messages required to infer the full state machine of the protocol.

To infer Mealy machines, we use Shahbaz and Groz's [104] variant of the classical L^* [2] inference algorithm. We describe only the intuition behind L^*, as the algorithm is well-described in the literature.

L^* is an online learning algorithm that proactively probes a black box with sequences of messages, listens to responses, and builds a finite state machine from the responses. The black box is expected to answer the queries in a faithful (i.e., it is not supposed to cheat) and deterministic way. Each generated sequence starts from the initial state, meaning that L^* has to reset the black box before sending each sequence. Once it converges, L^* conjectures a state machine, but it has no way to verify that it is equivalent to what the black box implements. Three approaches to solving this problem have been described in the literature. The first approach is to assume an existence of an *oracle* capable of answering the *equivalence queries*. L^* asks the oracle whether the conjectured state machine is equivalent to the one implemented by the black box, and the oracle responds either with 'yes' if the conjecture is equivalent, or with a counterexample, which L^* uses to refine the learned state machine and make another conjecture. The process is guaranteed to terminate in time polynomial in the number of states and the size of the input alphabet. However, in practice, such an oracle is unavailable. The second approach is to generate random *sampling queries* and use those to test the equivalence between the conjecture and the black box. If a sampling query discovers a mismatch between a conjecture and the black box, refinement is done the same way as with the counterexamples that would be generated by equivalence queries. The sampling approach provides a probabilistic guarantee [2] on the accuracy of the inferred state machine. The third approach, called black box model checking [96], uses bounded model checking to compare the conjecture with the black box. This approach requires that the input alphabet Σ_I of the checked system is known and its guarantees depend on the time and space resources spent. Out of the three options for checking conjectures, we chose to check conjectures using the sampling approach (the second approach).

As discussed in Section 5.3.1, MACE requires an output message abstraction function $\alpha_O : \mathcal{M}_O \rightarrow \Sigma_O$, where $\mathcal{M}_O$ is the set of all concrete output messages, to abstract concrete output messages into the abstract output alphabet. To simplify our notation, we overload the output abstraction function to operate on message sequences as follows:

Let $o = o_0 o_1 \cdots o_{n-1} \in \mathcal{M}_O^*$ be a sequence of concrete output messages. Its abstraction function $\alpha_O : \mathcal{M}_O^* \rightarrow \Sigma_O^*$ is defined as $\alpha_O(o) = \alpha_O(o_0)\alpha_O(o_1) \cdots \alpha_O(o_{n-1})$.

Unlike the prior work [27], MACE requires no input abstraction function.

5.4 Approach

We begin this section by a high-level description of MACE, illustrated in Figure 5.2. After the high-level description, each section describes a major component of MACE: abstract model inference, concrete state-space exploration, and filtering of redundant concrete input messages together with the abstract model refinement.

5.4.1 A High-Level Description

Suppose we want to infer a complete Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ representing some protocol, as implemented by the given program. We assume to know the output abstraction function α_O that abstracts concrete output messages into Σ_O. To bootstrap MACE, we also assume to have an initial set $\Sigma_{I0} \subseteq \Sigma_I$ of input messages, which can be extracted from a regression test suite, collected by observing the communication of the analyzed program with the environment, or obtained from DART and similar approaches [21, 22, 53, 102]. The initial Σ_{I0} alphabet could be empty, but MACE would take longer to converge. In our work, we used regression test suites provided with the analyzed applications, or extracted messages from a single observed communication session if the test suite was not available.

Next, we use Shahbaz and Groz's [104] variant of L^* algorithm to infer the first state machine $M_0 = (Q_0, \Sigma_{I0}, \Sigma_O, \delta_0, \lambda_0, q_0^0)$ with Σ_{I0} and Σ_O as the abstract alphabets. The L^* algorithm also maintains a data structure (called observation table [2]) that contains a set of shortest transfer sequences $q_0^0 \xrightarrow{s} q$, one for each inferred state $q \in Q_0$. We use such sequences to drive the program to one of the concrete states represented by the abstract state q. Since each abstract state could correspond to a large cluster of concrete states (Fig. 5.1), we use concolic execution to explore the clusters of concrete states around abstract states.

The state-space exploration generates sequences of concrete input and the corresponding output messages. Using the output abstraction function α_O, we can abstract the concrete output message sequences into sequences over Σ_O^*. However, we cannot abstract the concrete input messages into a subset of Σ_I, as we do not have the concrete input message abstraction function. Using all the concrete input messages for the L^*-based inference would be computationally infeasible. The state-space exploration discovers hundreds of thousands of concrete messages, because we run the exploration phase for hundreds of hours, and on average, it discovers several thousand new concrete messages per hour.

Thus, we need a way to filter out redundant messages and keep the ones that will allow L^* to discover new states. The filtering is done as follows. Suppose that $s = s_0 \cdots s_{n-1}$ is a sequence of concrete input messages generated from the exploration phase and $o \in \Sigma_O^*$ a sequence of the corresponding abstract output messages. We perform a brute-force search (over all permutations) for a sequence of abstract input messages $t \in \Sigma_{I0}^*$ such that M_0 accepts t generating o. If such a sequence is found, we discard s. Otherwise, we include to the refined input alphabet Σ_{I1} all concrete messages s_j of the sequence s, for any j such that $0 \leq j < n$, because at least one of these concrete messages can generate either a new state or a new transition.

With the new abstract input alphabet Σ_{I1}, we infer a new, more refined, abstract model M_1 and repeat the process. If the number of messages is finite and the exploration phase either terminates or runs for a predetermined bounded amount of time, MACE terminates as well.

5.4.2 Model Inference with L^*

MACE learns the abstract model of the analyzed program by constructing sequences of input messages, sending them to the program, and reasoning about the responses. For the inference, we use Shahbaz and Groz's [104] variant of L^* for learning Mealy machines. The inference process is similar as in our prior work [27].

In every iteration of MACE, L^* infers a new state machine over Σ_{Ii} and the new messages discovered by the state-space exploration guided by M_i, and conjectures M_{i+1}, a refinement of M_i. Out of the three options for checking conjectures discussed in Section 5.3.3, we chose to check conjectures using the sampling approach. We perform the check after $MACE$ completes all of its iterations, but in no experiment we performed did sampling discover any new states.

5.4.3 The State-Space Exploration Phase

We use the model inferred in Section 5.4.2 to guide the state-space exploration. For every state $q^i \in Q_i$ of the just inferred abstract model M_i, we compute a shortest transfer sequence of input messages from the initial state q^i_0. Suppose the computed sequence is $s \in \Sigma^*_{Ii}$. With $s = s_0 \cdots s_{n-1}$, we drive the analyzed application to a concrete state abstracted by the q^i state in the abstract model. All messages s_j, for any j such that $0 \leq j < n$, are concrete messages either from the set of seed messages, or generated by previous state-space exploration iterations. Thus, the process of driving the analyzed application to the state being analyzed consists of only computing a shortest path in M_i to the state, collecting the input messages along the path $q^i_0 \xrightarrow{+} q^i$, and feeding that sequence of concrete messages into the application.

Once the application reaches in the state being analyzed q^i, we run concolic execution from that state to explore the surrounding concrete states (Figure 5.1). In other words, the transfer sequence of input messages produces a concrete run, which is then followed by symbolic execution that computes the corresponding path-condition. Once the path-condition is computed, concolic execution resumes its normal exploration. We bound the time allotted to exploring the vicinity of every abstract state. In every iteration, we explore only the newly discovered states, i.e., $Q_i \backslash Q_{i-1}$. Re-exploring the same states over and over would be unproductive.

Thanks to the abstract model, MACE can easily compute the necessary input message permutations required to reach any abstract model state, just by computing a shortest path. On the other hand, approaches that combine concrete and symbolic execution have to negate multiple predicates and get the decision procedure to generate the required sequence of concrete input messages to get to a particular state. MACE has more control over this process, and our experimental results show that the increased control results in higher line coverage, deeper analysis, and more vulnerabilities found.

5.4.4 Model Refinement

The exploration phase described in Section 5.4.3 generates a large number (hundreds of thousands in our setting) of new concrete messages. Using all of them to refine the abstract model is both unrealistic, as inference is polynomial in the size of the alphabet, and redundant, as many messages are duplicates and belong to the same equivalence class. To reduce the number of input messages used for inference, Comparetti et al. [29] propose a message clustering technique. We take a different approach.

In the spirit of concolic execution, the exploration phase solves the path-condition (using a decision procedure) to generate new concrete inputs, more precisely, sequences of concrete input messages. During the concrete part of the exploration phase, such sequences of input messages are executed concretely, which generates the corresponding sequence of output messages. We abstract the generated sequence of output messages using α_O. If the abstracted sequence can be generated by the current abstract model, we discard the sequence; otherwise we add all the corresponding concrete input messages to Σ_{Ii}. We define this process more formally:

Definition 3 (Filter Function). *Let $\mathcal{M}_I$ (resp. $\mathcal{M}_O$) be a (possibly infinite) set of all possible concrete input (resp. output) messages. Let $s = s_0 \cdots s_{|s|-1} \in \mathcal{M}_I^*$ (resp. $o \in \mathcal{M}_O^*$) be a sequence of concrete input (resp. output) messages such that $|s| = |o|$. We assume that each input message s_j produces o_j as a response. Let $M_i \in \mathcal{A}$ be the abstract model inferred in the last iteration and $\mathcal{A}$ the universe of all possible Mealy machines. The filter function $f : \mathcal{A} \times \mathcal{M}_I^* \times \mathcal{M}_O^* \to 2^{\mathcal{M}_I}$ is defined as follows:*

$$f(M_i, s, o) = \begin{cases} \emptyset & \textit{if} \quad \exists t \in \Sigma_{Ii}^* \,.\, \lambda_i(t) = \alpha_O(o) \\ \{s_j \mid 0 \leq j < |s|\} & \qquad \textit{otherwise} \end{cases}$$

In practice, a single input message could produce either no response or multiple output messages. In the first case, our implementation generates an artificial no-response message, and in the second case, it picks the first produced output message. A more advanced implementation could infer a subsequential transducer [112], instead of a finite-state machine. A subsequential transducer can transduce a single input into multiple output messages.

Once the exploration phase is done, we apply the filter function to all newly found input and output sequences s^k and o^k, and refine the alphabet Σ_{Ii} by adding the messages returned by the filter function. More precisely:

$$\Sigma_{I(i+1)} \leftarrow \Sigma_{Ii} \cup \bigcup_k f(M_i, s^k, o^k)$$

In the next iteration, L^* learns a new model M_{i+1}, a refinement of M_i, over the refined alphabet $\Sigma_{I(i+1)}$.

5.5 Implementation

In this section, we describe our implementation of MACE. The L^* component sends queries to and collects responses from the analyzed server, and thus can be seen as a client sending queries to the server and listening to the corresponding responses. Section 5.5.1 explains this interaction in more detail. Section 5.5.2 surveys the main model inference optimizations, including parallelization, caching, and filtering. Finally, Section 5.5.3 discusses our implementation of the state-space exploration.

5.5.1 L^* as a Client

Our implementation of L^* infers the protocol state machine over the concrete input and abstract output messages. As a client, L^* first resets the server, by clearing its environment variables and resetting it to the initial state, and then sends the concrete input message sequences directly to the server.

Servers have a large degree of freedom in how quickly they want to reply to the queries, which introduces non-deterministic latency that we want to avoid. For one server application we analyzed (Vino), we had to slightly modify the server code to assure immediate response. We wrote wrappers around the `poll` and `read` system calls so the server immediately respond to the L^*'s queries, modifying eight lines of code in Vino. Without these modifications, our implementation would still work but take a longer time to run.

5.5.2 Model Inference Optimizations

We have implemented the L^* algorithm with distributed master-worker parallelization of queries. L^* runs in the master node, and distributes its queries among the worker nodes. The worker nodes compute the query responses, by sending the input sequences to the server, collecting and abstracting responses, and sending them back to L^*.

Since model refinement requires L^* to make repeated queries across iterations, we maintain a cache to avoid re-computing responses to the previously seen queries. L^* looks up the input in the cache before sending queries to worker nodes.

As L^*'s queries could trigger bugs in the server application, responses could be inconsistent. For example, if L^* emits two sequences of input messages, s and t, such that s is a prefix of t, then the response to s should be a prefix of the response to t. Before adding an input-output sequence pair to the cache, we check that all the prefixes are consistent with the newly added pair, and report a warning if they are inconsistent.

After each inference iteration, we analyze the state machine to find redundant messages (Definition 2) and discard them. This simple, but effective, optimization reduces the load on the subsequent MACE iterations. This optimization is especially important for inferring the initial state machine from the seed inputs.

5.5.3 State-Space Exploration

Our implementation of the state-space exploration consists of two components: a shortest transfer sequence generator and the state-space explorer. A shortest transfer sequence generator is implemented through a simple modification of the L^* algorithm. The algorithm maintains a data structure (called observation table [2]) that contains a set of shortest transfer sequences, one for each inferred state. We modify the algorithm to output this set together with the final model. MACE uses sequences from the set to launch and initialize state-space explorers.

We use BitFuzz, our concolic execution engine discussed in Section 2.4, as a state-space explorer. BitFuzz's execution monitor, which is an existing component called TEMU (discussed in Chapter 2.4, provides the capability to save and restore program snapshots. To perform model-assisted exploration from a desired state in the model, we first set the program state to the snapshot of the initial state. Then, we drive the program to the desired state using the corresponding shortest transfer sequence, and start concolic execution from that state.

In all our experiments, we used the snapshot capability to skip the server boot process. More precisely, we boot the server, make a snapshot, and run all the experiments on the snapshot. We do not report the code executed during the boot in the line coverage results.

5.6 Experimental Evaluation

To evaluate MACE, we infer server-side models of two widely deployed network protocols: Remote Framebuffer (RFB) and Server Message Block (SMB). The RFB protocol is widely used in remote desktop applications, including GNOME Vino and RealVNC[1]. Microsoft's SMB protocol provides file and printer sharing between Windows clients and servers. Although the SMB protocol is proprietary, it was reverse-engineered and re-implemented as an open-source system, called Samba. Samba allows interoperability between Windows and Unix/Linux-based systems. In our experiments, we use Vino 2.26.1 and Samba 3.3.4 as reference implementations to infer the protocol models of RFB and SMB respectively. We discuss the result of our model inference in Section 5.6.2.

[1]Vino is the default remote desktop application in GNOME distributions; RealVNC reports over 100 million downloads (http://www.realvnc.com).

Once we infer the protocol model from one reference implementation, we can use it to guide state-space exploration of other implementations of the same protocol. Using this approach, we analyze RealVNC 4.1.2 and Windows XP SMB, without re-inferring the protocol state machine.

MACE found a number of critical vulnerabilities, which we discuss in Section 5.6.3. In Section 5.6.4, we evaluate the effectiveness of MACE, by comparing it to the baseline state-space exploration component of MACE without guidance.

5.6.1 Experimental Setup

For our state-space exploration experiments, we used the DETER Security testbed [8] comprised of 3GHz Intel Xeon processors. For running L^* and the message filtering, we used a few slower 2.27GHz Intel Xeon machines. When comparing MACE against the baseline approach, we sum the inference and the state-space exploration time taken by MACE, and compare it to running the baseline approach for the same amount of time. This setup gives a slight advantage to the baseline approach because inference was done on slower machines, but our experiments still show MACE is significantly superior, in terms of achieved coverage, found vulnerabilities and exploration depth.

In addition to a subject binary program and a set of previously collected incoming packets, our implementation also requires knowledge of packet formats for message abstractions. In each of our experiments, this knowledge was obtained manually from the protocol specifications. The detail of each abstraction will be mentioned in Section 5.6.2.

5.6.2 Model Inference and Refinement

We used MACE to iteratively infer and refine the protocol models of RFB and SMB, using Vino 2.26.1 and Samba 3.3.4 as reference implementations respectively. Table 5.1 shows the results of iterative model inference and refinement on Vino and Samba.

As discussed in Section 5.4.2, once MACE terminates, we check the final inferred model with sampling queries. We used 1000 random sampling queries composed of 40 input messages each, and tried to refine the state machine beyond what MACE inferred. The sampling did not discover any new state in any experiment we performed.

Vino. For Vino, we collected a 45-second network trace of a remote desktop session, using `krdc` (KDE Remote Desktop Connection) as the client. During this session, the Vino server received a total of 659 incoming packets, which were considered as seed messages.

Program (Protocol)	Iter.	$\lvert Q \rvert$	$\lvert \Sigma_I \rvert$	$\lvert \Sigma_O \rvert$	Tot. Learning Time (min)
Vino	1st	7	8	7	142
(RFB)	2nd	7	12	8	8
Samba	1st	40	40	14	2028
(SMB)	2nd	84	54	24	1840
	3rd	84	55	25	307

Table 5.1: Model Inference Result at the End of Each Iteration. The second column identifies the inference iteration. The Q column denotes the number of states in the inferred model. The Σ_I (resp. Σ_O) column denotes the size of the input (resp. output) alphabet. The last column gives the total time (sum of all parallel jobs together) required for learning the model in each iteration, including the message filtering time. The learning process is incremental, so later iterations can take less time, as the older conjecture might need a small amount of refinement.

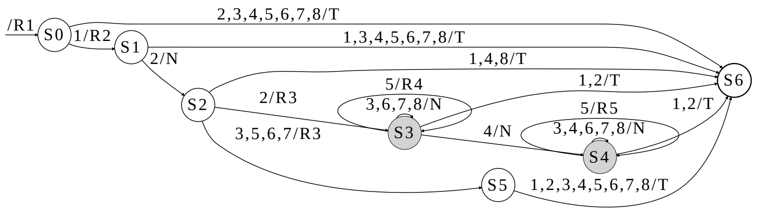
For abstracting the output messages, we used the message type and the encoding type of the outbound packets from the server. MACE inferred the initial model consisting of seven states, and filtered out all but 8 input and 7 output messages, as shown in Figure 5.3a.

Using the initial inferred RFB protocol model, the state-space explorer component of MACE discovered 4 new input messages and refined the model with new edges without adding new states (Figure 5.3b). We manually inspected the newly discovered output message (label R6 in Figure 5.3b) and found that it represents an outgoing message type not seen in the initial model.

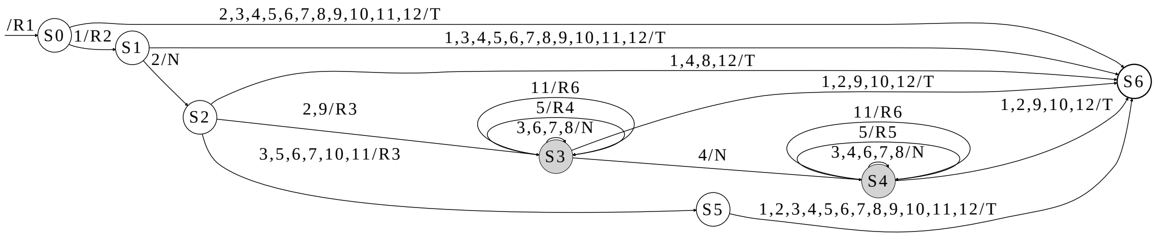
Since MACE found no new states that could be explored with the state-space explorer, the process terminated. Through manual comparison with the RFB protocol specification, we found that MACE has discovered all the input messages and all the states, except the states related to authentication and encryption, both of which we disabled in our experiments. Further, MACE found all the responses to client's queries[2].

We also performed an experiment with authentication enabled (encryption was still disabled). In this experiment, we used the first byte of each message for output abstraction because it corresponds to the location of the message type and the encoding type used in the experiment with authentication disabled. With this configuration, MACE discovered only three states, because it was not able to get past the checksum used during authen-

[2]There are two other output message types that are triggered by the server's GUI events and thus are outside of our scope.



(a) Original Vino's RFB Model Based on Observed Live Traffic.



(b) Final Vino's RFB Model Inferred by MACE.

Figure 5.3: Model Inference of Vino's RFB protocol. States in which MACE discovers vulnerabilities are shown in grey. The edge labels show the list of input messages and the corresponding output message separated by the '/' symbol. The explanations of the state and the input/output message encodings are in Figure 5.4.

tication, but discovered an infinite loop vulnerability that can be exploited for denial-of-service attacks. For this configuration to work, further integration of MACE and the decomposition and restitching technique, discussed in Chapter 4, or other techniques for reverse-engineering of message encryption [18, 115] are required.

Samba. For Samba, we collected a network trace of multiple SMB sessions, using Samba's `gentest` test suite[3], which generates random SMB operations for testing SMB servers. We used the default `gentest` configuration, with the default random number generator seeds. To abstract the outbound messages from the server, we used the SMB message type and status code fields; error messages were abstracted into a single error message type. The Samba server received a total of 115 input messages, from which MACE inferred an initial SMB model with 40 states, with 40 input and 14 output messages (after filtering out redundant messages). Figure 5.5a shows the initial model.

In the second iteration, MACE discovered 14 new input and 10 new output messages and refined the initial model from 40 states to 84 states. The model converged in the third iteration after adding a new input and a new output message without adding new states.

[3]http://samba.org/∼tridge/samba_testing/

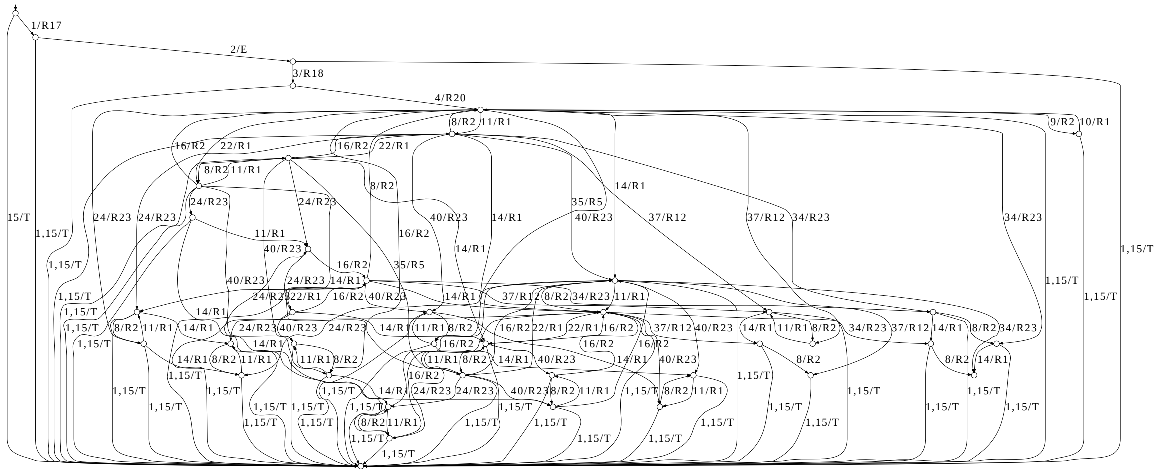
Label	Description
1	client’s protocol version
2	byte 0x01 (securityType=None, clientInit)
3	setPixelFormat message
4	setEncodings message
5	frameBufferUpdateRequest message
6	keyEvent message
7	pointer event message
8	clientCutText message
9	byte 0x22
10	malformed client’s protocol version
11	frameBufferUpdateRequest message with bpp=8 and true-color=false
12	malformed client’s protocol version

(a) Input Legend.

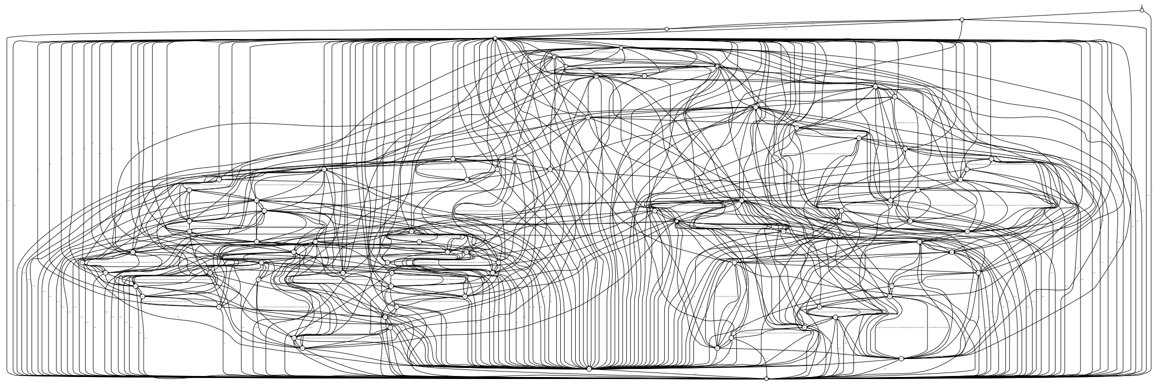
Label	Description
R1	server’s protocol version
R2	server’s supported security types
R3	serverInit message
R4	framebufferUpdate message with default encoding
R5	framebufferUpdate message with alternative encoding
R6	setColourMapEntries message
N	no explicit reply from server
T	socket closed by server

(b) Output Legend.

Figure 5.4: Explanation of States and Input/Output Messages of the State Machine from Figure 5.3.



(a) The Initial SMB Model Inferred from the Seed Messages.



(b) Converged SMB Model.

Figure 5.5: The Inferred SMB Model from Samba.

Table 5.1 summarizes all three inference rounds. The converged model is depicted in Figure 5.5b.

Manually analyzing the inferred state machine, we found that some of the discovered input messages have the same type, but different parameters, and therefore have different effects on the server (and different roles in the protocol). MACE discovered all the 67 message types used in Samba, but the concrete messages generated by the decision procedure during the state-space exploration phase often had invalid message parameters, so the server would simply respond with an error. Such responses do not refine the model and are filtered out during model inference. In total, MACE was successful at generate valid messages with 23 (out of 67) message types, which is an improvement over 13 message types exercised by the test suite.

Label	Desc.	Label	Desc.	Label	Desc.	Label	Desc.
1	negprot	15	invalid	29	checkpath	43	fclose
2	sesssetupX	16	rmdir	30	mkdir	44	ulogoffX
3	sesssetupX	17	readX	31	mv	45	fclose
4	tconX	18	lseek	32	open	46	trans
5	unlink	19	close	33	open	47	tdis
6	trans2	20	ntrename	34	ntcreateX	48	findnclose
7	trans2	21	openX	35	mv	49	dskattr
8	rmdir	22	mkdir	36	trans2	50	findclose
9	rmdir	23	ntcreateX	37	openX	51	exit
10	mkdir	24	ntcreateX	38	trans2	52	dskattr
11	mkdir	25	trans2	39	setatr	53	ctemp
12	tconX	26	trans2	40	ntcreateX	54	getatr
13	nttrans	27	lockingX	41	dskattr	55	create
14	mkdir	28	writeX	42	fclose		

(a) Input Legend.

Label	Desc.	Label	Desc.	Label	Desc.
R1	mkdir_success	R10	exit_success	R19	ulogoffX_success
R2	rmdir_success	R11	trans_success	R20	tconX_success
R3	open_success	R12	openX_success	R21	dskattr_success
R4	create_success	R13	trans2_success	R22	fclose_success
R5	mv_success	R14	findclose_success	R23	ntcreateX_success
R6	getatr_success	R15	findnclose_success	E	error
R7	setatr_success	R16	tdis_success	T	session terminated by server
R8	ctemp_success	R17	negprot_success		
R9	checkpath_success	R18	sesssetupX_success		

(b) Output Legend.

Figure 5.6: Explanation of Input/Output Messages of the State Machine from Figure 5.5.

We identified several causes of incompleteness in message discovery. First, message validity is configuration dependent. For example, the `spoolopen`, `spoolwrite`, `spoolclose` and `spoolreturnqueue` message types need an attached printer to be deemed valid. Our experimental setup did not emulate the complete environment, precluding us from discovering some message types. Second, a single `echo` message type generated by MACE induced the server to behave inconsistently and we discarded it due to our determinism requirement. Although this is likely a bug in Samba, this behavior is not reliably reproducible. We exclude this potential bug from the vulnerability reports that we provide later. Third, our infrastructure is unable to analyze the system calls and other code executed in the kernel space. In effect, the computed symbolic constraints are underconstrained. Thus, some corner-cases, like a specific combination of the message type and parameter (e.g., a specific file name), might be difficult to generate. This is a general problem when the symbolic formula computed by symbolic execution is underconstrainted.

In our experiments, we used Samba's default configuration, in which encryption is disabled. The SMB protocol allows null-authentication sessions with empty password, similar to anonymous FTP.

MACE converged relatively quickly in both Vino and Samba experiments (in three iterations or less). We attribute this mainly to the granularity of abstraction. A finer-grained model would require more rounds to infer. The granularity of abstraction is determined by the output abstraction function, (Section 5.3.1).

5.6.3 Discovered Vulnerabilities

We use the inferred models to guide the state-space exploration of implementations of the inferred protocol. After each inference iteration, we count the number of newly discovered states, generate shortest transfer sequences (Section 5.3.3) for those states, initialize the server with a shortest transfer sequence to the desired (newly discovered) state, and then run 2.5 hours of state-space exploration in parallel for each newly discovered state. The input messages discovered during those 2.5 hours of state-space exploration per state are then filtered and used for refining the model (Section 5.4.4). For the baseline concolic execution without model guidance, we run $|Q|$ parallel jobs with different random seeds for each job for 15 hours, where $|Q|$ is the number of states in the final converged model inferred for the target protocol. Different random seeds are important, as they assure that each baseline job explores different trajectories within the program.

We rely upon the operating system runtime error detection to detect vulnerabilities, but other detectors, like Valgrind[4], could be used as well. Once MACE detects a vulnerability, it generates an input sequence required for reproducing the problem. When analyzing Linux applications, MACE reports a vulnerability when any of the critical exceptions (`SIGILL`, `SIGTRAP`, `SIGBUS`, `SIGFPE`, and `SIGSEGV`) is detected. For Windows programs, a vulnerability is found when MACE traps a call to `ntdll.dll::KiUserExceptionDispatcher` and the value of the first function argument represents one of the critical exception codes.

MACE found a total of seven vulnerabilities in Vino 2.26.1, RealVNC 4.1.2, and Samba 3.3.4, within 2.5 hours of state-space exploration per state. In comparison, the baseline concolic execution without model-guidance, found only one of those vulnerabilities (the least critical one), even when given the equivalent of 15 hours per state. Four of the vulnerabilities MACE found are new and also present in the latest version of the software at the time of writing. The list of vulnerabilities is shown in Table 5.2. The rest of this section provides a brief description of each vulnerability.

Vino. MACE found three vulnerabilities in Vino; all of them are new. The first one (CVE-2011-0904) is an out-of-bounds read from arbitrary memory locations. When a certain type of the RFB message is received, the Vino server parses the message and later uses two of the message value fields to compute an unsanitized array index to read from. A remote attacker can craft a malicious RFB message with a very large value for one of the fields and exploit a target host running Vino. The Gnome project labeled this vulnerability with the "Blocker" severity (bug 641802), which is the highest severity in their ranking system, meaning that it must be fixed in the next release. MACE found this vulnerability after 122 minutes of exploration per state, in the first iteration (when the inferred state machine has seven states, Table 5.1). The second vulnerability (CVE-2011-0905) is an out-of-bounds read due to a similar usage of unsanitized array indices; the Gnome project labeled this vulnerability (bug 641803) as "Critical", the second highest problem severity. This vulnerability is marked as a duplicate of CVE-2011-0904, for it can be fixed by patching the same point in the code. However, these two vulnerabilities are reached through different paths in the finite-state machine model and the out-of-bounds read happens in different functions. These two vulnerabilities are actually located in a library used by not only Vino, but also a few other programs. According to Debian security tracker[5], kdenetwork 4:3.5.10-2 is also vulnerable.

The third vulnerability (CVE-2011-0906) is an infinite loop, found in the configuration with authentication enabled. The problem appears when the Vino server receives an

[4]http://valgrind.org/
[5]http://security-tracker.debian.org/tracker/CVE-2011-0904

authentication input from the client larger than the authentication checksum length that it expects. When the authentication fails, the server closes the client connection, but leaves the remaining data in the input buffer queue. It also enters a deferred-authentication state where all subsequent data from the client is ignored. This causes an infinite loop where the server keeps receiving callbacks to process inputs that it does not process in deferred-authentication state. The server gets stuck in the infinite loop and stops responding, so we classify this vulnerability as a denial-of-service vulnerability. Unlike all other discovered vulnerabilities, we discovered this one when L^* hung, rather than by catching signals or trapping the exception dispatcher. Currently, we have no way of detecting this vulnerability with the baseline, so we do not report the baseline results for CVE-2011-0906.

Samba. MACE found 3 vulnerabilities in Samba. The first two vulnerabilities have been previously reported and are fixed in the latest version of Samba. One of them (CVE-2010-1642) is an out-of-bounds read caused by the usage of an unsanitized Security_Blob_Length field in SMB's Session_Setup_AndX message. The other (CVE-2010-2063) is caused by the usage of an unsanitized field in the "Extra byte parameters" part of an SMB Logoff_AndX message. The third one is a null pointer dereference caused by an unsanitized Byte_Count field in the Session_Setup_AndX request message of the SMB protocol. To the best of our knowledge, this vulnerability has never been publicly reported but has been fixed in the latest release of Samba. We did not know about any of these vulnerabilities prior to our experiments.

RealVNC. MACE found a new critical out-of-bounds write vulnerability in RealVNC. One type of the RFB message processed by RealVNC contains a length field. The RealVNC server parses the message and uses the length field as an index to access the process memory without performing any sanitization, causing an out-of-bounds write.

Win XP SMB. The implementation of Win SMB is partially embedded into the kernel, and currently our concolic execution system does not handle the kernel operating system mode. Thus, we were able to explore only the user-space components that participate in handling SMB requests. Further, we found that many involved components seem to serve multiple purposes, not only handling SMB requests, which makes their exploration more difficult. We found no vulnerabilities in Win XP SMB.

Program	Vulnerability Type	Disclosure ID	Iter.	Jobs ($\|Q\|$)	Search Time			
					MACE		Baseline	
					per job (min)	total (hrs)	per job (min)	total (hrs)
Vino	Wild read (blocker)	*CVE-2011-0904*	1/2	7	122	15	$>$900	$>$105
	Out-of-bounds read	*CVE-2011-0905*	1/2	7	31	4	$>$900	$>$105
	Infinite loop	*CVE-2011-0906*†	1/2	7	1	1	N/A	N/A
Samba	Buffer overflow	CVE-2010-2063	1/3	84	88	124	$>$900	$>$1260
	Out-of-bounds read	CVE-2010-1642	1/3	84	10	14	$>$900	$>$1260
	Null-ptr dereference	Fixed w/o CVE	1/3	84	8	12	430	602
RealVNC	Out-of-bounds write	*CVE-2011-0907*	1/1	7	17	2	$>$900	$>$105
Win XP SMB	None	None	None	84	$>$150	$>$210	$>$900	$>$1260

Table 5.2: Description of the Found Vulnerabilities. The upper half of the table (Vino and Samba) contains results for the reference implementations from which the protocol model was inferred, while the bottom half (Real VNC and Win XP SMB) contains the results for the other implementations that were explored using the inferred model (from Vino and Samba). The disclosure column lists Common Vulnerabilities and Exposures (CVE) numbers assigned to vulnerabilities MACE found. The new vulnerabilities are *italicized*. The † symbol denotes a vulnerability that could not have been detected by the baseline approach, because it lacks a detector that would register non-termination. We found it with MACE, because it caused L^* to hang. The "Iter." column lists the iteration in which the vulnerability was found and the total number of iterations. The "Jobs" column contains the total number of parallel state-space exploration jobs. The number of jobs is equal to the number of states in the final converged inferred state machine. The baseline experiment was done with the same number of jobs running in parallel as the MACE experiment. The MACE column shows how much time passed before at least one parallel state-space exploration job reported the vulnerability and the total runtime (number of jobs $\times$ time to the first report) of all the jobs up to that point. The "Baseline" column shows runtimes for the baseline concolic execution without model guidance. We set the timeout for the MACE experiment to 2.5 hours per job. The baseline approach found only one vulnerability, even when allowed to run for 15 hours (per job). The $> t$ entries mean that the vulnerability was not found within time t.

Program (Protocol)	Sequential Time (min)	Instruction Coverage			Total crashes (Unique locations)	
		Baseline	MACE	improvement	Baseline	MACE
Vino (RFB)	1200	129762	138232	6.53%	0 (0)	2 (2)
Samba (SMB)	16775	66693	105946	58.86%	20 (1)	21 (5)
RealVNC (RFB)	1200	39300	47557	21.01%	0 (0)	7 (2)
Win XP (SMB)†	16775	90431	112820	24.76%	0 (0)	0 (0)

Table 5.3: Instruction Coverage Results. The table shows the instruction coverage (number of unique executed instruction addresses) of MACE after 2.5 hours of exploration per state in the final converged inferred state machine, and the baseline concolic execution given the amount of time equivalent to (time MACE required for inferring the final state machine + number of states in the final state machine × 2.5 hours), shown in the second column. For example, from Table 5.1, we can see that Samba inference took the total of $2028 + 1840 + 307 = 4175$ minutes and produced an 84-state model. Thus, the baseline approach was given $84 \times 150 + 4175 = 16775$ minutes to run. The last two columns show the total number of crashes each approach found, and the number of unique crash locations (EIPs) in parenthesis. Due to a limitation of our implementation of the state-space exploration (user-mode only), the baseline result for Windows XP SMB (marked †) was so abysmal, that comparing to the baseline would be unfair. Thus, we compute the Win XP SMB baseline coverage by running Samba's `gentest` test suite.

5.6.4 Comparison with the Baseline

We ran several experiments to illustrate the improvement of MACE over the baseline concolic execution approach. First, we measured the instruction coverage of MACE on the analyzed programs and compared it against the baseline coverage. Second, we compared the number of crashes detected by MACE and by the baseline approach over the same amount of time. This number provides an indication of how diverse the execution paths discovered by each approach are: more crashes imply more diverse searched paths. Finally, we compared the effectiveness of MACE and the baseline approach to reach deep states in the final inferred model.

Instruction Coverage. In this experiment, we measured the numbers of unique instruction addresses (i.e., EIP values) of the program binary and its libraries covered by MACE and the baseline approach. These numbers show how effective the approaches are at uncovering new code regions in the analyzed program. For Vino, RealVNC, and Samba, we used concolic execution as the baseline approach and ran the experiment using the setup outlined in Section 5.6.1. We ran MACE allowing 2.5 hours of state-space exploration per each inferred state. To provide a fair comparison, we ran the baseline for the amount of time that is equal to the sum of the MACE's inference and state-space exploration times. As shown in Table 5.3, our result illustrates that MACE provides a significant improvement in the instruction coverage over concolic execution.

As mentioned before, our tool currently works on user-space programs only. Because Windows SMB is mostly implemented as a part of the Windows kernel, the results of the baseline approach were abysmal. To avoid a straw man comparison, we chose to compare against Samba's `gentest` test suite, regularly used by Samba developers to test the SMB protocol. Using the test suite, we generate test sequences and measure the obtained coverage. As for other experiments, we allocated the same amount of time to both the test suite and MACE. The experimental results clearly show MACE's ability to augment test suites manually written by developers.

Number of Detected Crashes. Using the same setup as in the previous experiment, we measured the number of crashing input sequences generated by each approach. We report the number of crashes and the number of unique crash locations. From each category of unique crash locations, we manually processed the first four reported crashes. All the found vulnerabilities (Table 5.2) were found by processing the very first crash in each category. All the later crashes we processed were just variants of the first reported crash. MACE found 30 crashing input sequences with 9 of them having unique crash locations

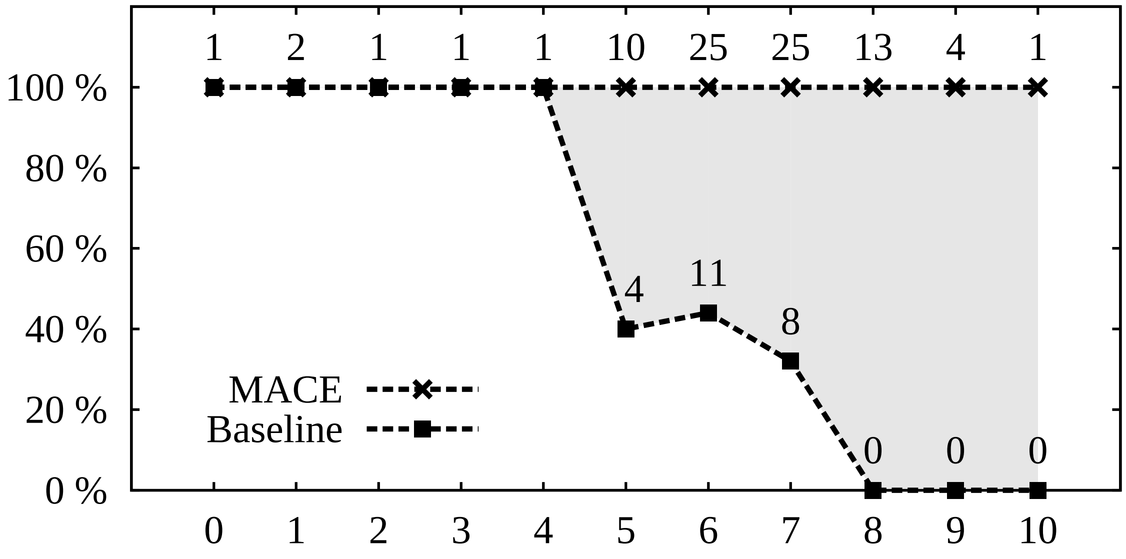


Figure 5.7: SMB Exploration Depth. The inferred state machine can be seen as a directed graph. Suppose we compute a spanning tree (e.g., [30]) of that graph. The root of the graph is at level zero. Its children are at level one, and so on. The figure shows the percentage of states visited at each level by MACE and the baseline approach. The numbers above points show the number of visited states at the given depth. The shaded area clearly shows that MACE is superior to the baseline approach in reaching deep states of the inferred protocol.

(the EIP of the crashed instruction). In comparison, the baseline approach only found 20 crashing input sequences, all of them having the same crash location.

Exploration Depth. Using the same setup as for the coverage experiment, we measured how effective each approach is in reaching deep states. The inferred state machine can be seen as a directed graph. Suppose we compute a spanning tree (e.g., [30]) of that graph. The root of the graph is at level zero. Its children are at level one, and so on. We measured the percentage of states reached at every level. Figure 5.7 clearly shows that MACE is superior to the baseline approach in reaching deep states in the inferred protocol.

5.7 Limitations

Completeness is a problem for any dynamic analysis technique. Accordingly, MACE cannot guarantee that all the protocol states will be discovered. Incompleteness stems from the following: (1) each state-space explorer instance runs for a bounded amount of time and some inputs may simply not be discovered before the timeout, (2) among multiple shortest transfer sequences to the same abstract state, MACE picks one, potentially missing further exploration of alternative paths, (3) similarly, among multiple concrete input messages with the same abstract behavior, MACE picks one and considers the rest redundant (Definition 2).

Our approach to model inference and refinement is not entirely automatic: the end users need to provide an abstraction function that abstracts concrete output messages into an abstract alphabet. Coming up with a good output abstraction function can be a difficult task. If the provided abstraction is too fine-grained, model inference may be too expensive to compute or may not even converge. On the other hand, the inferred model may fail to distinguish two interesting states if the abstraction is too coarse-grained. Nevertheless, our approach provides an important improvement over our prior work [28], which requires abstraction functions for both input and output messages.

When using our approach to learn a model of a proprietary protocol, a certain level of protocol reverse-engineering is required prior to running MACE. First, we need a basic level of understanding of the protocol interface to be able to correctly replay input messages to the analyzed program. For example, this may require overwriting the cookie or session-id field of input messages so that the sequence appears indistinguishable from real inputs to the target program. Second, our approach requires an appropriate output abstraction, which in turn requires understanding of the output message formats. Message format reverse-engineering is an active area of research [20, 35, 36] out of the scope of our study.

Encryption is a difficult problem for every (existing) protocol inference technique. To circumvent the issue, we configure the analyzed programs not to use encryption. However, for proprietary protocols, such a configuration may not be available and a further integration of MACE and the decomposition and restitching technique, discussed in Chapter 4, or other techniques for reverse-engineering of message encryption [18, 115] are required.

5.8 Conclusion

We have proposed MACE, a new approach to software state-space exploration. MACE iteratively infers and refines an abstract model of the protocol, as implemented by the program, and exploits the model to explore the program's state-space more effectively. By applying MACE to four server applications, we show that MACE (1) improves coverage up to 58.86%, (2) discovers significantly more vulnerabilities (seven vs. one), and (3) performs significantly deeper search than the baseline approach.

We believe that further research is needed along several directions. First, a deeper analysis of the correspondence of the inferred finite state models to the structure and state-space of the analyzed application could reveal how models could be used even more effectively than what we propose here. Second, it is an open question whether one could design effective automatic abstractions of the concrete input messages. The filtering function we propose here is clearly effective, but might drop important messages. Third, the finite-state models might not be expressive enough for all types of applications. For example, subsequential transducers [112] might be the next, slightly more expressive, representation that would enable us to model protocols more precisely, without significantly increasing the inference cost. Fourth, MACE currently does no white box analysis, besides concolic execution for discovering new concrete input messages. MACE could also monitor the value of program variables, consider them as the input and the output of the analyzed program, and automatically learn the high-level model of the program's state-space. This extension would allow us to apply MACE to more general classes of programs.

Chapter 6

Conclusion

6.1 Potential Integration of the Proposed Techniques

The scaling techniques presented in this thesis could be integrated but such integration has not materialized. The level of re-engineering effort required is a major reason we decided not to go on with the integration. This section will discuss the issue as well as other challenges that would arise when integrating our scaling techniques together.

Currently, our implementation — specifically its legacy component called TEMU (discussed in Section 2.4) — could reliably save a program state only once; further saves tend to damage the TEMU disk images we used for the experiments. TEMU is developed as an extension of QEMU version 0.9.1 [97], which it inherits this reliability issue from. Since the time of TEMU implementation, QEMU has been re-engineered from the ground up to resolve various issues including the aforementioned reliability issue, and the older version, from which TEMU is based on, is no longer supported. Because the new version of QEMU is significantly different from the old version, TEMU has to be reimplemented if it were to support the new version of QEMU and to resolve the reliability issue. This reimplementation has not completely materialized.

One requirement for a successful integration of our techniques is the ability to save and restore the program state efficiently. If we were to combine stitched concolic execution with the other two techniques, we need to be able to handle cases when there are multiple instances of serial decomposition (e.g., multiple encrypted input fields controlled by a loop for LECE and sequences of encrypted message for MACE). In such a case, our implementation must perform re-stitching at the entry of each encoding functions, in the reverse order. The ability to save and restore the program state efficiently at each function entry is required. This means that the lengthy reimplementation of TEMU is needed and thus we decided not to go on with the integration.

Integrating LECE and stitched concolic execution would require further assumptions and heuristics to link the techniques together. Considering the case in which a loop operates on a variable-length input field inside a decompression/decryption function, LECE alone would try to generate constraints based on the loop and the function input but stitched concolic execution would try to avoid solving such constraints. As a result, we need to link the LECE-generated constraints to other constraints outside the encoding function. For example, we might use program analysis to compose a new constraint based on the relationship between the length of the input and the output of a decompression/decryption function (e.g., that they are of the same length), and conjoin it with the LECE-generated constraints. The conjoined constraint would be in terms of the function output (instead of the function input) and stitched concolic execution can pass it to the decision procedure.

Integrating LECE into MACE's state-space exploration step (Section 5.4.3) would improve the performance of MACE further. Instead of spending much time inside loops, state-space exploration using LECE would focus on the overall effects of the loops and could explore more states. The new input messages generated during each concolic execution step would be of variable length instead of fixed length. However, if not handled properly, the generation of variable length input messages would slow down MACE significantly when complex long messages are repeatedly fed back to MACE later on. To avoid such a setback, we have to tweak the decision procedure so that, whenever possible, it preferably generates short messages rather than the longer ones.

6.2 Conclusion

Concolic execution of program code is important for security-related applications. However, basic implementations of concolic execution only work well on certain classes of programs, such as commercial software and malware for which source code is not available. In this thesis, we have developed scalable techniques that extend symbolic reasoning to more classes of binary programs. We have demonstrated that our scaling techniques significantly speed up the process of automatic test input generation, which is the most common application of concolic execution. We have also shown that our techniques enable some of these previously unexplored applications, such as malware genealogy and protocol model inference, which were hindered by the scalability issue of concolic execution.

Poor handling of loops is a known issue of the traditional approach of concolic execution. The approach is limited to examining behavior of a program one execution path at a time and thus becomes susceptible to the combinatorial explosion in the number of feasible execution paths which is prominent with the existence of loops. In this thesis, we have developed a new scaling technique, loop-extended concolic execution, which provides a middle ground for handling loops. Through automatic software vulnerability discovery, we have confirmed that our scaling technique significantly reduces the number of program executions required to discover buffer overflow bugs. We have also shown that loop-extended concolic execution allows us to describe vulnerability conditions in term of loop-related properties and lengths of input fields and helps improve defense against future attacks of known vulnerabilities.

Data decryption, data encryption, and the computation of checksums and hash functions are difficult to reason about automatically. Concolic execution naturally has issues when a program under analysis contains such functions. To address this issue, we have developed a technique that scales concolic execution to reason about programs that use encoding functions. The technique is based on decomposing the formulas induced by a program, solving only a subset, and then re-stitching the solutions into a complete result. Through automatic malware vulnerability discovery, we have shown that our scaling techniques improve the speed and reduces the memory usage of a concolic execution engine, making it more practical. Using the vulnerabilities we found, we have surveyed and confirmed our hypothesis that there are components in malware which tend to evolve slowly over time and thus could be used to identify the malware family an unknown suspicious binary belongs to.

To help scale concolic execution to large network applications that communicate with their environment through some protocols, we have developed an iterative process of combining concolic execution with knowledge of an abstract model of the program under analysis. Our technique can iteratively infer and refine an abstract model that represents the high-level logic of the network applications being analyzed. Through vulnerability discovery, we have illustrated that our combined technique performs faster and deeper program analysis than the traditional approach.

Altogether, we have shown that concolic execution techniques can be scaled to broad classes of programs and are useful in a variety of important security applications. We hope that the study presented in this thesis will encourage further research of applying concolic execution and other related techniques to any security-related issues that may arise in the near future.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, second edition, 2006. 33, 44

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. 12, 85, 88, 89, 91, 92, 93, 98

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*, pages 283–300, Feb. 2011. 2, 16

[4] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, Apr. 2004. 43

[5] G. Balakrishnan and T. W. Reps. DIVINE: DIscovering Variables IN Executables. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Jan. 2007. 43

[6] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI'01: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, 2001. 87

[7] M. Barnett, R. Deline, M. Fähndrich, B. Jacobs, K. R. Leino, W. Schulte, and H. Venter. Verified software: Theories, tools, experiments. chapter The Spec# Programming System: Challenges and Directions, pages 144–152. Springer-Verlag, 2008. 86

[8] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Design, deployment, and use of the DETER testbed. In *Proc. of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test*. USENIX Association, 2007. 23, 99

[9] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983. 15

[10] BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`. 43, 56

[11] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Mar. 2009. 41, 44

[12] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium*, pages 213–228, Berkeley, California, USA, Aug. 2007. USENIX Association. 2, 16, 20

[13] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, May 2006. 2, 4, 16, 19, 26, 53

[14] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Computer Security Foundations*, July 2007. 53

[15] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008. 23

[16] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, pages 391–408, Reston, VA, USA, Mar. 2010. Internet Society. 68, 69, 74, 82

[17] J. Caballero, Z. Liang, P. Poosankam, and D. Song. Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration. In *RAID'09: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, Heidelberg, Germany, Sept. 2009. Springer. 2, 3, 4, 7, 16, 19, 31, 38, 43, 52, 83

[18] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *CCS'09: Proceedings of the 16th ACM Conference on Computer and Communications Security*,

pages 621–634, New York, NY, USA, Nov. 2009. ACM. 2, 16, 20, 71, 77, 82, 87, 101, 112

[19] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communication Security*, Chicago, IL, October 2010. vi

[20] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *CCS'07: Proc. of the 14th ACM Conf. on Computer and Communications Security*, pages 317–329. ACM, 2007. 20, 31, 38, 51, 87, 112

[21] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, Berkeley, California, USA, Dec. 2008. USENIX Association. 1, 15, 93

[22] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN'05: Proceedings of the 12th International SPIN Workshop on Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23, Heidelberg, Germany, Aug. 2005. Springer. 1, 15, 41, 93

[23] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Computer and Communications Security*, Nov. 2006. 1

[24] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA'08: Proceedings of the Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 143–163, Heidelberg, Germany, July 2008. Springer. 82

[25] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proc. of the IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2012. 16

[26] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011. vi

[27] C. Y. Cho, D. Babić, R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *CCS'10: Proc. of the 2010 ACM Conf. on Computer and Communications Security*, pages 426–440. ACM, 2010. 12, 87, 90, 93, 94

[28] C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song. Insights from the inside: A view of botnet management from infiltration. In *LEET'10: Proc. of the 3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–1. USENIX Association, 2010. 87, 112

[29] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *S&P'09: Proc. of the 2009 30th IEEE Symposium on Security and Privacy*, pages 110–125. IEEE Computer Society, 2009. 86, 87, 95

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001. xiii, 111

[31] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Symposium on Operating Systems Principles*, Oct. 2007. 26, 48, 53

[32] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Symposium on Operating Systems Principles*, Oct. 2005. 26, 52

[33] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, Jan. 1978. 52

[34] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. In *ACM Transactions on Architecture and Code Optimization*, pages 359–389, Dec. 2006. 52

[35] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. of 16th USENIX Security Symposium*, pages 1–14. USENIX Association, 2007. 20, 87, 112

[36] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irún-Briz. Tupni: Automatic reverse engineering of input formats. In *CCS'08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 391–402. ACM, 2008. 20, 87, 112

[37] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy*, May 2007. 7, 26, 50, 53

[38] CVE: Common vulnerabilities and exposures. `http://cve.mitre.org/`. 78

[39] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, Heidelberg, Germany, Mar. 2002. 57

[40] D. Danchev. Help! someone hijacked my 100k+ Zeus botnet!, Feb. 2009. `http://ddanchev.blogspot.com/2009/02/help-someone-hijacked-my-100k-zeus.html`. 16, 80

[41] D. De, A. Kumarasubramanian, and R. Venkatesan. Inversion attacks on secure hash functions using SAT solvers. In *SAT'07: Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 377–382, Heidelberg, Germany, 2007. Springer. 59

[42] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010. 86

[43] D. Dittrich, F. Leder, and T. Werner. A case study in ethical decision making regarding remote mitigation of botnets. In *WECSR'10: Workshop on Ethics in Computer Security Research*, Lecture Notes in Computer Science, Heidelberg, Germany, Jan. 2010. Springer. 80

[44] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Programming Language Design and Implementation*, June 2003. 52

[45] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984. 56

[46] N. Falliere and E. Chien. Zeus: King of the bots, 2009. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/zeus_king_of_bots.pdf`. 71

[47] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Computer and Communications Security*, Oct. 2003. 26, 52

[48] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07: Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531, Heidelberg, Germany, July 2007. Springer. 21, 40, 44

[49] V. Ganesh, T. Leek, and M. C. Rinard. Taint-based directed whitebox fuzzing. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 474–484, Washington, DC, USA, May 2009. IEEE Computer Society. 15, 56, 82

[50] ghttpd. `http://gaztek.sf.net/ghttpd/`. 48

[51] P. Godefroid. Compositional dynamic test generation. In *POPL'07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, New York, NY, USA, Jan. 2007. ACM. 52, 83

[52] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI'08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 206–215, June 2008. 31, 41, 44, 52, 83

[53] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI'05: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, June 2005. ACM. 1, 15, 17, 41, 85, 93

[54] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS'08: Proceedings of the Network and Distributed System Security Symposium*, Reston, VA, USA, Feb. 2008. The Internet Society. 1, 15, 23, 26, 52

[55] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *FSE'06: Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 117–127. ACM, 2006. 88

[56] S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *Principles of Programming Languages*, Jan. 2003. 52

[57] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, K. Fu, and D. Song. Take two software updates and see me in the morning: The case for software security evaluations of medical devices. In *Proceedings of the 2nd USENIX Workshop on Health Security and Privacy (HealthSec)*, San Francisco, CA, Aug. 2011. 16

[58] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *SPIN'03: Proc. of the 10th Int. Workshop on Model Checking of Software*, volume 2648 of *LNCS*, pages 235–239. Springer-Verlag, 2003. 87

[59] P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD'00: Proc. of the 2000 IEEE/ACM Int. Conf. on Computer-aided design*, pages 120–126. IEEE Press, 2000. 88

[60] IDA Pro. `http://www.hex-rays.com/idapro/`. 43

[61] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical Report (Computer Science) No. 32, Bell Laboratories, July 1975. 38

[62] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976. 52

[63] M. Kassner. The top 10 spam botnets: New and improved, Feb. 2010. `http://blogs.techrepublic.com.com/10things/?p=1373`. 71

[64] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, Washington, DC, USA, May 2009. IEEE Computer Society. 2, 16

[65] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. 15

[66] M. Kobayashi. Dynamic characteristics of loops. *IEEE Transactions on Computers*, 33(2):125–132, Feb. 1984. 32, 33

[67] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In *SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, Washington, DC, USA, May 2010. IEEE Computer Society. 81, 82

[68] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for Honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference*, Dec. 2005. 20

[69] J. Leyden. Monster botnet held 800,000 people's details. *The Register*, Mar. 2010. `http://www.theregister.co.uk/2010/03/04/mariposa_police_hunt_more_botherders/`. 57

[70] Z. Li, M. Sanghi, Y. Chen, Ming-Yang Kao, and B. Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006. 19

[71] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security*, Feb. 2008. 31

[72] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Symposium on Applied Computing*, Mar. 2008. 52

[73] M86 Security Labs. Botnet statistics for week ending April 11, 2010, Apr. 2010. `http://www.m86security.com/labs/bot_statistics.asp`. 71

[74] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, Nov. 2007. 31, 52

[75] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI'08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 193–205, New York, NY, USA, June 2008. ACM. 68

[76] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. 90

[77] Smashing the Mega-D/Ozdok botnet in 24 hours. `http://blog.fireeye.com/research/2009/11/smashing-the-ozdok.html`. 16, 81

[78] Microsoft Corporation. Microsoft security bulletin MS07-046: Vulnerability in GDI could allow remote code execution, Aug. 2007. 48

[79] Microsoft Corporation. *SQL Server Resolution Protocol Specification*, Jan. 2009. Revision 0.6.1. 49

[80] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990. 15

[81] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, Mar. 2006. 52

[82] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th USENIX Security Symposium*, pages 67–81, Berkeley, California, USA, Aug. 2009. USENIX Association. 15

[83] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July-Aug. 2003. 48

[84] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997. 32, 43

[85] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Transactions on Programming Languages and Systems*, 29(5), Aug. 2007. 52

[86] National Institute of Standards and Technology, Gaithersburg, MD, USA. *Federal Information Processing Standard 180-2: Secure Hash Standard*, Aug. 2002. 55, 57

[87] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005. 19

[88] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security*, Feb. 2005. 52

[89] OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org/`. 68, 69

[90] OSVDB. Cutwail Bot svchost.exe CC Message Handling Remote Overflow, July 2010. `http://osvdb.org/66497`. 75

[91] OSVDB. Gheg Bot RtlAllocateHeap Function Null Dereference Remote DoS, July 2010. `http://osvdb.org/66498`. 75

[92] OSVDB. Zbot Trojan svchost.exe Compressed Input Handling Remote Overflow, July 2010. `http://osvdb.org/66501`. 75

[93] OSVDB. Zbot Trojan svchost.exe Network Message Crafted Payload Size Handling Infinite Loop Remote DoS, July 2010. `http://osvdb.org/66500`. 75

[94] OSVDB. Zbot Trojan svchost.exe RtlAllocateHeap Function Null Dereference Remote DoS, July 2010. `http://osvdb.org/66499`. 75

[95] W. A. Owens, K. W. Dam, and H. S. Lin, editors. *Technology, Policy, Law, and Ethics Regarding U.S. Acquisition and Use of Cyberattack Capabilities*. The National Academies Press, Washington, DC, USA, 2009. 80

[96] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *Proc. of the IFIP TC6 WG6.1 Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 225–240. Kluwer, B.V., 1999. 92

[97] QEMU. `http://wiki.qemu.org/Main_Page`. 115

[98] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving progrms. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, September 2010. 20

[99] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA'09: Proceedings of the 18th ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 225–236, New York, NY, USA, July 2009. ACM. vi

[100] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint tracking. In *Code Generation and Optimization*, Apr. 2008. 43

[101] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, May 2010. 17

[102] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference held jointly with Foundations of Software Engineering*, Sept. 2005. 15, 41, 93

[103] Shadowserver foundation. `http://www.shadowserver.org/`. 81

[104] M. Shahbaz and R. Groz. Inferring Mealy machines. In *FM'09: Proc. of the 2nd World Congress on Formal Methods*, pages 207–222. Springer, 2009. 92, 93, 94

[105] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis (keynote invited paper). In *ICISS'08: Proceedings of*

the 4th International Conference on Information Systems Security, volume 5352 of *Lecture Notes in Computer Science*, pages 1–25, Heidelberg, Germany, Dec. 2008. Springer. 16, 21, 43, 56, 69

[106] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007. 56

[107] Cutwails poorly written code leads to heavy SSL traffic, Feb. 2010. `http://blog.threatfire.com/2010/02/page/2`. 71

[108] J. Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley, 2005. 15

[109] N. Tillmann and J. De Halleux. Pex: white box test generation for .net. In *TAP'08: Proceedings of the 2nd international conference on Tests and proofs*, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag. 1

[110] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Formal methods and testing. chapter Model-based testing of object-oriented reactive systems with spec explorer, pages 39–76. Springer-Verlag, 2008. 86

[111] S. Venkataraman, A. Blum, and D. Song. Limits of learning-based signature generation with adversaries. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, Feb. 2008. 19

[112] J. M. Vilar. Query learning of subsequential transducers. In *Proc. of the 3rd Int. Colloquium on Grammatical Inference: Learning Syntax from Sentences*, pages 72–83. Springer-Verlag, 1996. 96, 113

[113] T. Wang, TaoWei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS'09: Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, CA, 2009. 15

[114] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, Washington, DC, USA, May 2010. IEEE Computer Society. 8, 82

[115] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic reverse engineering of encrypted messages. In *ESORICS'09: 14th European Symposium*

on Research in Computer Security, volume 5789 of *Lecture Notes in Computer Science*, pages 200–215, Heidelberg, Germany, Sept. 2009. Springer. 8, 82, 87, 101, 112

[116] Wireshark. `http://www.wireshark.org`. 44

[117] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Network and Distributed System Security*, Feb. 2008. 20, 31

[118] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, New York, NY, USA, Jan. 2005. ACM. 83

[119] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Foundations of Software Engineering held jointly with European Software Engineering Conference*, Sept. 2003. 26, 52

[120] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *International Symposium on Software Testing and Analysis*, July 2008. 26, 45, 47, 52

[121] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD'01: Proc. of the Int. Conf. on Computer-Aided Design*, pages 279–285. IEEE Press, 2001. 85

[122] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Foundations of Software Engineering*, Nov. 2004. 6, 26, 45, 52

[123] The zlib library. `http://www.zlib.net/`. 68